R2P2: Making RPCs first-class datacenter citizens

Marios Kogias, George Prekas, Adrien Ghosn, Jonas Fietz, and Edouard Bugnion, *EPFL*

https://www.usenix.org/conference/atc19/presentation/kogias-r2p2

This paper is included in the Proceedings of the 2019 USENIX Annual Technical Conference.

July 10–12, 2019 • Renton, WA, USA

ISBN 978-1-939133-03-8

Open access to the Proceedings of the 2019 USENIX Annual Technical Conference is sponsored by USENIX.

R2P2: Making RPCs first-class datacenter citizens

Marios Kogias George Prekas Adrien Ghosn Jonas Fietz Edouard Bugnion

EPFL, Switzerland

Abstract

Remote Procedure Calls are widely used to connect datacenter applications with strict tail-latency service level objectives in the scale of µs. Existing solutions utilize streaming or datagram-based transport protocols for RPCs that impose overheads and limit the design flexibility. Our work exposes the RPC abstraction to the endpoints and the network, making RPCs first-class datacenter citizens and allowing for in-network RPC scheduling.

We propose R2P2, a UDP-based transport protocol specifically designed for RPCs inside a datacenter. R2P2 exposes pairs of requests and responses and allows efficient and scalable RPC routing by separating the RPC target selection from request and reply streaming. Leveraging R2P2, we implement a novel *join-bounded-shortest-queue (JBSQ)* RPC load balancing policy, which lowers tail latency by centralizing pending RPCs in the router and ensures that requests are only routed to servers with a bounded number of outstanding requests. The R2P2 router logic can be implemented either in a software middlebox or within a P4 switch ASIC pipeline.

Our evaluation, using a range of microbenchmarks, shows that the protocol is suitable for µs-scale RPCs and that its tail latency outperforms both random selection and classic `HTTP` reverse proxies. The P4-based implementation of R2P2 on a Tofino ASIC adds less than 1µs of latency whereas the software middlebox implementation adds 5µs latency and requires only two CPU cores to route RPCs at 10 Gbps line-rate. R2P2 improves the tail latency of web index searching on a cluster of 16 workers operating at 50% of capacity by 5.7× over NGINX. R2P2 improves the throughput of the Redis key-value store on a 4-node cluster with master/slave replication for a tail-latency service-level objective of 200µs by more than 4.8× vs. vanilla Redis.

1 Introduction

Web-scale online data-intensive applications such as search, e-commerce, and social applications rely on the scale-out architectures of modern, warehouse-scale datacenters to meet service-level objectives (SLO) [7, 17]. In such deployments, a single application can comprise hundreds of software components, deployed on thousands of servers organized in multiple tiers and connected by commodity Ethernet switches. The typical pattern for web-scale applications distributes the critical data (*e.g.,* the social graph) in the memory of hundreds of data services, such as memory-resident transactional databases [26, 85, 87–89], NoSQL databases [62, 78], key-value stores [22,54,59,67,93], or specialized graph stores [14]. Consequently, online data-intensive (OLDI) applications are deployed as 2-tier applications with root servers handling end-user queries and leaf servers holding replicated, sharded data [8, 58]. This leads to a high fan-in, high fan-out connection graph between the tiers of an application that internally communicates using RPCs [11]. Each client must (a) fan-out an RPC to the different shards and (b) within each shard, choose a server from among the replica set. Moreover, each individual task can require from only few microseconds (µs) of user-level execution time for simple key-value requests [54] to a handful of milliseconds for search applications [35].

To communicate between the tiers, applications most commonly layer RPCs on top of TCP, either through RPC frameworks (*e.g.,* gRPC [31] and Thrift [86]) or through application-specific protocols (*e.g.,* Memcached [59]). This leads to a mismatch between TCP, which is a byte-oriented, streaming transport protocol, and message-oriented RPCs. This mismatch introduces several challenges, one of which is RPC load distribution. In one approach, root nodes randomly select leaves via direct connections or L4-load balancing. This approach leads to high fan-in, high fan-out communication patterns, load-imbalance and head-of-line blocking. The second approach uses a L7 load balancer or reverse proxy [1, 16, 25] to select among replicas on a per request basis, *e.g.,* using a Round-Robin or Join-Shortest-Queue (JSQ) algorithm. While such load balancing policies improve upon random selection, they do not eliminate head-of-line blocking. Furthermore, the load balancer can become a scalability bottleneck.

This work proposes a new communication abstraction for datacenter applications that exposes RPCs as first-class citi-

zens of the datacenter not only at the client and server endpoints, but also in the network. Endpoints have direct control over RPC semantics, do not suffer from head-of-line blocking because of connection multiplexing, and can limit buffering at the endpoints. The design also enables RPC-level processing capabilities for in-network software or hardware middleboxes, including scheduling, load-balancing, straggler-mitigation, consensus and in-network aggregation.

As a first use case, we show how to use our network protocol to implement efficient, scalable, tail-tolerant, high-throughput routing of RPCs. Our design includes an RPC router that can be implemented efficiently either in software or within a programmable switch ASIC such as P4 [12]. In addition to classic load balancing policies, we support *Join-Bounded-Shortest-Queue* (`JBSQ(n)`), a new RPC scheduling policy that splits queues between the router and the servers, allowing only a bounded number of outstanding requests per server, which significantly improves tail-latency.

We make the following contributions :

- The design of *Request-Response Pair Protocol (R2P2)*, a transport protocol designed for datacenter µs-RPCs that exposes the RPC abstraction to the network and the endpoints, breaks the point-to-point RPC communication assumptions, and separates request selection from message streaming, nearly eliminating head-of-line blocking.
- The implementation of the R2P2 router on a software middlebox that adds only 5µs to end-to-end unloaded latency and is capable of load balancing incoming RPCs at line rate using only 2 cores.
- The implementation of the R2P2 router within a P4-programmable Tofino dataplane ASIC, which eliminates the I/O bottlenecks of a software middlebox and reduces latency overhead to 1µs.
- The implementation of `JBSQ(n)`, a split-queue scheduling policy that utilizes a single in-network queue and bounded server queues and improves tail-latency even for µs-scale tasks.

Our evaluation with microbenchmarks shows that our R2P2 deployment with a `JBSQ(3)` router achieves close to the theoretical optimal throughput for 10µs tasks across different service time distributions for a tail-latency SLO of 150µs and 64 independent workers. Running Lucene++ [56], an open-source websearch library over R2P2, shows that R2P2 outperforms conventional load balancers even for coarser-grain, millisecond-scale tasks. Specifically, R2P2 lowers the 99^{th} percentile latency by 5.7× at 50% system load over NGINX with 16 workers. Finally, running Redis [78], a popular key-value store with built-in master-slave replication, over R2P2 demonstrates an increase of 4.8×– 5.6× in throughput vs. vanilla Redis (over TCP) at a 200µs tail-latency SLO for different `read:write` ratios. The Redis improvements are due to the cumulative benefits of a leaner protocol, kernel bypass, and scheduling improvements.

The paper is organized as follows: §2 provides the necessary background. §3 describes the R2P2 protocol and §4 its implementation. §5 is the experimental evaluation of R2P2. We discuss related work in §6 and conclude in §7. The R2P2 source code is available at `https://github.com/epfl-dcsl/r2p2`.

2 Background

2.1 Datacenter RPCs

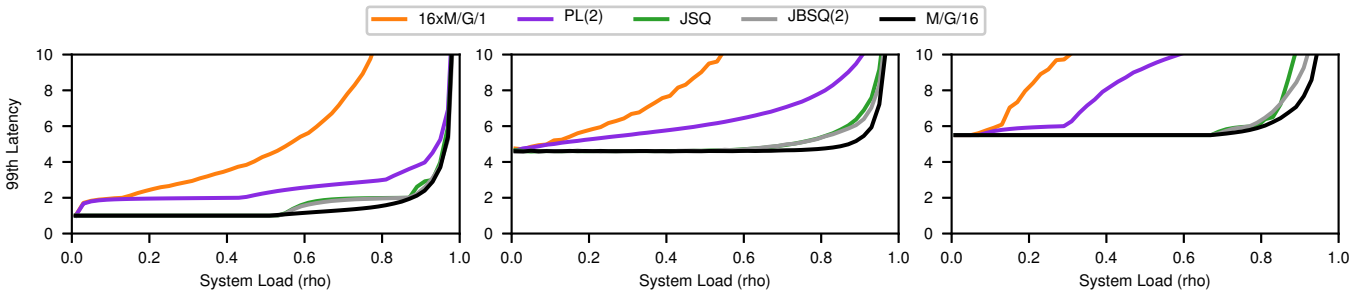
TCP has emerged as the main transport protocol for latency-sensitive, intra-datacenter RPCs running on commodity hardware, as its reliable stream semantics provide a convenient abstraction to build upon. Such use is quite a deviation from the original design of a wide-area, connection-oriented protocol for both interactive (*e.g.,* `telnet`) and file transfer applications. TCP's generality comes with a certain cost as RPC workloads usually consist of short flows in each direction. In many cases, the requests and replies are small and can fit in a single packet [5, 63]. Although RDMA is an alternative, it has specific hardware requirements and can be cumbersome to program, leading to application-specific solutions [22, 42, 43].

Overall, the requirements of RPCs differ from the assumptions made by TCP in terms of failure semantics, connection multiplexing, API scalability, and end-point buffering:

RPC semantics: Some datacenter applications choose weak consistency models [18] to lower tail latency. These applications typically decompose the problem into a series of independent, often idempotent, RPCs with no specific ordering guarantees. Requests and responses always come in pairs that are semantically independent from other pairs. Thus, the reliable, in-order stream provided by TCP far stronger than the applications needs and comes with additional network and system overheads.

Connection multiplexing: To amortize the setup cost of TCP flows, RPCs are typically layered on top of persistent connections, and most higher-level protocols support multiple outstanding RPCs on a flow, *e.g.,* `HTTP/2`, memcache, *etc.* Multiplexing different RPCs on the same flow implies ordering the requests that share a socket, even though the individual RPCs are semantically independent. This ordering limits scheduling choices and can lead to Head-of-Line-Blocking (HOL). HOL appears when fast requests are stuck behind a slower request and when a single packet drop affects multiple pending requests.

Connection scalability: The high fan-in, high fan-out patterns of datacenter applications lead to large number of connections and push commodity operating systems beyond their



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

Figure 1: Simulation results for the 99^{th} percentile latency across 3 service time distributions with $\bar{S} = 1$

efficiency point. Recent work has addressed the issue either by deviating from the POSIX socket interface while maintaining TCP as the transport [9] or by developing custom protocols, *e.g.,* to deploy memcached on a combination of connection-less UDP for RPC `get` and router proxy for RPC `set` [67].

Endpoint bufferbloat: Prior work has addressed network-specific issues of congestion management and reliability within the network [2, 3]. Unfortunately, the use of TCP via the POSIX socket API leads to buffering in both endpoints over which applications have little control or visibility [45]. Applications willing to trade-off harvest vs. yield [29] would ideally never issue RPCs with no chance of returning by the deadline due to buffering in the network stack.

2.2 Load balancing

The problem of spreading out load extends to load balancing across servers within a distributed, scale-out environment. Load balancers encapsulate a set of servers behind a single virtual IP address and improve the availability and capacity of applications. Load balancing decisions, however, can severely affect throughput and tail-latency; thus, a significant amount of infrastructure is dedicated to load balancing [23, 60]. Load balancers can be implemented either in software [23, 64, 69] or in hardware [1, 16, 25, 60] and fall into two broad categories: (1) Layer-4 ("network") load balancers that use the 5-tuple information of the TCP or UDP flow to select a destination server. The assignment is static and independent of the load; (2) Layer-7 ("application") load balancers come in the form of `HTTP` reverse proxies as well as protocol-specific routers implemented in software middleboxes [67] or SDN switches [13, 15]. These load balancers terminate the client TCP connections, use dynamic policies to select a target, and reissue the request to the server on a different connection.

Layer-7 load balancers support many policies to decide the eventual RPC target, including random, power-of-two [61], round-robin, Join-Shortest-Queue (JSQ), and Join-Idle-Queue (JIQ) [55]. Layer-7 load balancers are ubiquitous at the web tier and can theoretically mitigate tail-latency better, due to their dynamic policies. However, they are less commonly deployed within tiers of applications to support µs-scale RPCs. The reasons for this are (i) the increased latency due to the extra hop (ii) the scalability issues introduced when all requests and responses flow through a proxy.

2.3 As a queuing theory problem

In this section, we approach the problem of RPC load balancing from a theoretical point of view by abstracting away system aspects using basic queuing theory. We show the benefits of request-level load balancing over random-selection among distributed queues (which is equivalent to L4 load balancing) in improving tail-latency, and we evaluate different request-level load balancing policies.

Fortunately, the theoretical answers are clear: single-queue, multi-worker models (*i.e.,* `M/G/k` according to Kendall's notation) perform better than distributed multi-queue models (*i.e.,* `k×M/G/1`, with one queue per worker) because they are work-conserving and guarantee that requests are processed in order [49, 90].

Between those two extremes, there are other models that improve upon random selection and are practically implementable through L7 load balancing. Power-of-two [61] (`PL(2)`), or similar schemes, are still in the realm of randomized load balancing, but perform better than a blind random selection. `JSQ` performs close to a single queue model for low-variability service times [55].

We define Join-Bounded-Shortest-Queue `JBSQ(n)` as a policy that splits queues between a centralized component with an unbounded queue and distributed bounded queues of maximum depth `n` for each worker (including the task currently processed). The single-queue model is equivalent to `JBSQ(1)` whereas `JSQ` is equivalent to `JBSQ(`∞`)`.

Figure 1 quantifies the tail-latency benefit, at the 99^{th} percentile, for these queuing models observed in a discrete event simulation. We evaluate a configuration with a Poisson arrival process, $k = 16$ workers, and three well-known distributions with the same service time $\bar{S} = 1$. These distributions are: deterministic, exponential and bimodal-1 (where 90% of re-

quests execute in .5 and 10% in 5.5 units) [55].

From the simulation results, we conclude that: (1) there is a dramatic gap in performance between the random, multi-queue model and the single-queue approach, which is optimal among FCFS queuing systems. (There is no universally optimal scheduling strategy for tail-latency [90].) (2) `PL(2)` improves upon random selection, but these benefits diminish as service time variability increases. `JSQ` performs close to the optimal for low service time variability. (3) `JBSQ(2)`, while it deviates from the single queue model, outperforms `JSQ` under high load as the service time variability increases.

These results are purely theoretical and in particular assume perfect global knowledge by the scheduler or load balancer. This global view would be the result of communication between the workers and the load balancer in a real deployment. Any practical system must consider I/O bottlenecks and additional scheduling delays because of this communication. In this paper, we make the claim that `JBSQ(n)` can be implemented in a practical system and can deliver maximal throughput with small values of n even for µs-scale tasks, thus minimizing tail latency and head-of-line blocking.

3 R2P2: A transport protocol for RPCs

We propose R2P2 (*Request-Response Pair Protocol*), a UDP-based transport protocol specifically targeting latency-critical RPCs within a distributed infrastructure, *i.e.,* a datacenter. R2P2 exposes the RPC abstraction to the network, thus allowing for efficient in-network request-level load balancing.

R2P2 is a connectionless transport protocol capable of supporting higher-level protocols such as `HTTP` without protocol-level modifications. Unlike traditional multiplexing of the RPC onto a reliable byte-oriented connection, R2P2 is an inherently request/reply-oriented protocol that maintains no state across requests. The R2P2 request-response pair is initiated by the client and is uniquely identified by a triplet of $< src\_IP, src\_port, req\_id >$. This design choice decouples the request destination (set by the client) from the actual server that will reply, thus breaking the point-to-point RPC communication semantics and enabling the implementation of any request-level load balancing policy.

Figure 2 describes the interactions and the packets exchanged in sending and receiving an RPC within a distributed infrastructure that uses a request router to load balance requests across the servers. We illustrate the general case of a multi-packet request and a multi-packet response.

1. A `REQ0` message opens the RPC interaction, uniquely defined by the combination of source IP, UDP port, and an RPC sequence number. The datagram may contain the beginning of the RPC request itself.

2. The router identifies a suitable target server and directs the message to it. If there is no available server, requests can temporarily queue up in the router.

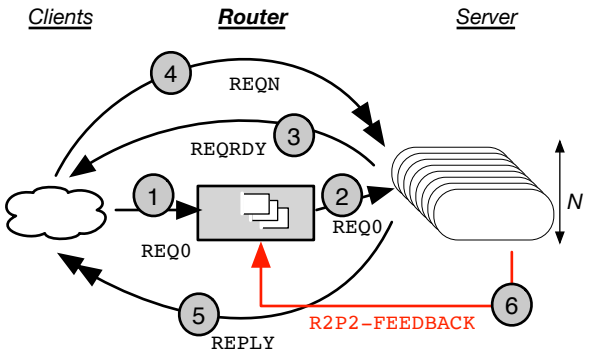


Figure 2: The R2P2 protocol for a request-reply exchange. Each message is carried within a UDP packet. Single arrows represent a single packet whereas double arrows represent a stream of datagrams.

3. If the RPC request exceeds the size of data in the `REQ0` payload, then the server uses a `REQready` message to inform the client that it has been selected and that it will process the request.

4. Following (3), the client directly sends the remainder of the request as `REQn` messages.

5. The server replies directly to the client with a stream of `REPLY` messages.

6. The servers send `R2P2-FEEDBACK` messages to the router to signal idleness, availability, or health, depending on the load balancing policies.

We note a few obvious consequences and benefits of the design: (i) Given that an RPC is identified by the triplet, responses can arrive from a different machine than the original destination. Responses are sent directly to the client, bypassing the router; (ii) there is no head-of-line blocking resulting from multiplexing RPCs on a socket, since there are no sockets and each request-response pair is treated independently; (iii) there are no ordering guarantees across RPCs; (iv) the protocol is suited for both short and long RPCs. By avoiding the router for `REQn` message and replies, the router capacity is only limited by its hardware packet processing rate, not by the overall amount of size of the messages.

Unlike protocols that blindly provide reliable message delivery, R2P2 exposes failures and delays to the application. R2P2 follows the end-to-end argument in systems design [80]. A client application initiates a request-response pair and determines the failure policy of each RPC according to its specific needs and SLOs. By propagating failures to the application, the developer is free to choose between *at-least-once* and *at-most-once* semantics by re-issuing the same request that failed. Unlike TCP, failures affect only the RPC in question, not other requests. This is useful in cases with fan-out replicated

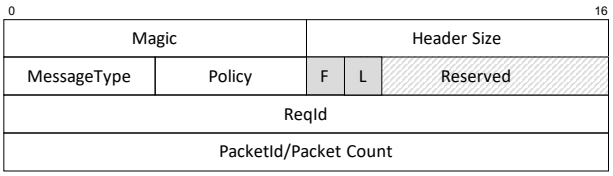


Figure 3: R2P2 Header Format

requests, where R2P2 can provide system support for the implementation of tail-mitigation techniques, such as hedged requests [17].

While novel in the context of µs-scale, in-memory computing, the connection "pair" is similar in spirit to the "exchange" that is the core of the SCSI/Fibre Channel protocol (FCP [27]). For example, a single-packet-request-multi-packet-response RPC over R2P2 would be similar to SCSI `read` within a single fibre channel exchange. Equivalently, an R2P2 multi-packet-request-single-packet-response would be similar to a SCSI `write`.

3.1 Transport considerations

Figure 3 describes a proposed R2P2 header, while Table 1 includes the different R2P2 messages. All R2P2 messages are UDP datagrams. R2P2 supports a 16-bit request id whose scope is local to the `(src_ip, src_port)` pair. As such, each client (`(src_ip, src_port)` pair) can have up to 65536 outstanding RPCs, well beyond any practical limitations. The R2P2 header also includes a 16-bit packet id meaning that each R2P2 message can consist of up to 65536 MTU-sized packets. The above two fields can be extended, if necessary, without changing the protocol semantics. Currently R2P2 uses two flags (`F, L`) to denote the first and last packet of a request.

Finally, the R2P2 header contains a `Policy` field, which allows client applications to directly specify certain policies to the router, or any other intermediate middlebox, for this specific RPC. Currently, the only implemented policies are `unrestricted`, which allows the router to direct `REQ0` packet to any worker in the set, and `sticky`, which forces the router to direct the message to the master worker among the set. This mechanism is central to our implementation of a tail-tolerant Redis, based on a master/slave architecture. It is used to direct writes to the master, but balances reads according to the load balancing policy. Additional policies, *e.g.,* session-stickiness, or policies implementing different consistency models, can be implemented in R2P2 middleboxes and will be identified by this header field, thus showcasing the benefits of R2P2's in-network RPC awareness.

Deployment assumptions: We assume that R2P2 is deployed within a datacenter, *i.e.,* the clients, router and servers are connected by a high-bandwidth, low-latency Ethernet fabric. We make no assumptions about the core network that can depend either on ECMP flow hashing or packet spraying [30, 32, 63]. R2P2 tolerates packet reordering within the same message and reconstructs the message at the end-point. By design, though, there is no ordering guarantee across RPCs, even if they are sent by the same client.

Message	Description
REQUEST	A message carrying an RPC request
REPLY	A message carrying an RPC reply
REQRDY	Sent by the server to the client to ack the REQ0 of a multi-packet request
R2P2-FEEDBACK	Sent by the server to the router
DROP	Sent by the router or the server to a client to explicitly drop a request
SACK	Sent by the client or the server to ask for missing packets in a request or reply

Table 1: The R2P2 message types

Timer management: Given that the assumed deployment model allows for packet reordering, packet loss detection depends on timers. There is one retransmission timeout `RTO` timer used for multi-packet requests or responses. It is in the order of milliseconds and triggers the transmission of a `SACK` message request for the missing packets. Servers garbage collect RPCs with failed multi-packet requests or multi-packet replies after a few `RTO`s. On the client side there is a timer set by the client application when sending the request. This timer is disarmed when the whole reply is received, and can be as aggressive as the application SLO. Based on this timer applications can implement tail-mitigation techniques [17] or early drop requests based on their importance.

Congestion management: R2P2 focuses on reducing queuing on the server side; we do not make any explicit contribution in congestion control. Instead, R2P2 can utilize existing solutions for congestion control, including (1) Homa [63], whose message semantics easily map to R2P2's request-response semantics and (2) ECN-based schemes such as DCTCP [2] and DCQCN [94]. Congestion control will be necessary only for multi-packet requests and replies (`REQN` and `REPLY`), and is independent of the interactions described in Fig 2.

Flow Control: R2P2 implements two levels of flow control, one between the client and the middlebox and one between the middlebox and the servers. R2P2 middleboxes can drop individual requests, either randomly or based on certain priority policies, if they become congested, without affecting other requests, thus implementing the first level of flow control. Based on the functionality and the policy, the middlebox is in charge of implementing the second level of flow control to the servers. In the `JBSQ` case, `JBSQ` limits the number of

outstanding requests on each server, thus servers can not be overwhelmed.

3.2 API

R2P2 exposes a non-POSIX API specifically designed for RPC workloads. Making RPCs first class citizens and exposing the request-response abstraction through the networking stack significantly simplifies writing client-server applications. Application code that traditionally implements the RPC logic on top of a byte stream abstraction is now part of the R2P2 layer of the networking stack.

Table 2 summarizes the corresponding application calls and callbacks for the client and server application. The API has an asynchronous design that allows applications to easily send and receive independent RPCs. When calling `r2p2_send_req` the client application sets the timer timeout and callback functions independently for each RPC request. The client and server applications are notified only when the entire response or request messages have arrived through the `req_success` and `req_recv` callbacks, equivalently.

3.3 JBSQ router design considerations

R2P2 exposes the request-response abstraction to the network as a first-class citizen. It is expected that a software or hardware middlebox will manipulate client requests to implement a certain policy, *e.g.,* scheduling, load balancing, admission control, or even application logic, *e.g.,* routing requests to the right server in a distributed hash table. In this section, we discuss the design choices regarding an R2P2 request router implementing the `JBSQ` scheduling policy. Similar ideas can be applied to other middleboxes with alternative functionality.

The choice of `JBSQ`: As seen in § 2.3 `JSQ` and `JBSQ` perform closer to the optimal single queue model. `JBSQ` though offers several practical benefits over `JSQ`. It implements router-servers flow control and can be implemented within a Tofino ASIC. JSQ requires finding the minimum among a number of values, which is hard to implement in a hardware dataplane. Also, JBSQ achieves better latency under high load and service time dispersion. That is because JSQ uses the queue size as a proxy for queuing time, which can be misleading in the presence of service-time dispersion.

Application Calls	
Type	**Description**
r2p2_poll	Poll for incoming req/resp
r2p2_send_req	Send a request
r2p2_send_response	Send a response
r2p2_message_done	Deallocate a request or response

Callbacks	
Type	**Description**
req_recv	Received a new request
req_success	Request was successful
req_timeout	Timer expired
req_error	Error condition

Table 2: The `r2p2-lib` API

R2P2-FEEDBACK messages: To implement the `JBSQ(n)` policy we leverage the `R2P2-FEEDBACK` messages provided by the R2P2 specification. These messages, sent by the servers back to the router after completing the service of a request, specify: (i) The maximum number of outstanding RPCs the server is willing to serve (the "`n`" in `JBSQ(n)`). By sending the current "`n`" in every `R2P2-FEEDBACK` message, servers can dynamically change the number of outstanding requests based on the application SLOs. (ii) The number of requests this server has served including the last request. The router uses this information to track the current number of outstanding requests in the server's bounded queue. This field makes the message itself idempotent and the protocol robust to `R2P2-FEEDBACK` drops.

We note that this approach puts each server in charge of controlling its own lifecycle by sending unsolicited `R2P2-FEEDBACK` messages, *e.g.,* to join a load balancing group, leave it, adjust its bounded queue size based on its idle time, or to periodically signal its idleness.

Direct client request - direct server return: R2P2 implements direct server return (DSR) [34, 65] since the replies do not go through the router. This is a widely-used technique in L4 load balancers with static policies [65]. R2P2 uses DSR while implementing request-level load balancing. In addition, R2P2 implements direct client request, where the router handles only the first packet of a multi-packet request, while the rest is streamed directly to the corresponding server, thus avoiding router IO bottlenecks.

Deployment: A software R2P2 router is deployed as a middlebox and traffic is directed to its IP address. The hardware R2P2 router is also deployed as an IP-addressed middlebox. The same hardware can also be a Top-of-Rack switch serving traffic to servers within the rack, following a "rack-scale" deployment pattern. In such a pattern, the router has full visibility on the RPC traffic to the rack and all packets go through the ToR switch. This could enable simplifications to the packet exchange, *e.g.,* using `R2P2-FEEDBACK` messages only for changing the depth of the bounded queues; the ToR can estimate their current size by tracking the traffic.

Router high availability: The router itself is nearly stateless and a highly-available implementation of the router is relatively trivial. Upon a router failure, only soft state regarding the routing policy is lost, including the current size of the per-worker bounded queue and the queue of pending RPCs. Clients simply failover to the backup router using a virtual IP address and reissue RPCs upon timeout, using the

exact same mechanism used to handle a REQ0 packet loss. Servers reconstruct the relationship with the router with their R2P2-FEEDBACK message to the new router.

Server membership: Servers behind the R2P2 router can fail and new servers can join the load balancing group. R2P2-FEEDBACK messages implicitly confirm to the router that a server is alive. In case of a failure, the lack of R2P2-FEEDBACK messages will prevent the router from sending requests to the failed server, and the bounded nature of JBSQ(n) limits the number of affected RPCs. Similarly, newly-added servers can send R2P2-FEEDBACK messages to the router informing about their availability to serve requests.

The choice of JBSQ(n): The choice of n in JBSQ is crucial. A small n will behave closer to a single-queue model, but will restrict throughput. The rationale behind the choice of n is similar to the Bandwidth Delay Product. On each queue there should be enough outstanding requests so that the server does not stay idle during the server-router communication. For example, for a communication delay of around 15 µs and a fixed service time of 10 µs, n=3 is enough to achieve full throughput. Shorter service times will require higher n values. High service time dispersion and batching on the server will also require higher values than what predicted by the heuristic. Servers can even dynamically adjust the value of n based on their processing rate and minimal idle time between requests.

4 Implementation

We implement (1) r2p2-lib as userspace Linux library on top of either UDP sockets or DPDK [21] (§4.1); (2) the software R2P2 router on top of DPDK (§4.2) and (3) the hardware solution in the $P4_{14}$ programming language [72] to run within a Barefoot Tofino ASIC [6] (§4.3).

4.1 r2p2-lib

The library links into both client and server application code. It exposes the previously described API and abstracts the differences between the Linux socket and the DPDK-based implementations. The current implementation is non-blocking and rpc_poll is typically called in a spin loop. To do so, we depend on epoll for Linux, while for DPDK we implemented a thin ARP, IP, and UDP layer on top of DPDK's polling mode driver, and exposed that to r2p2-lib. Our C implementation of r2p2-lib consists of 1300 SLOC.

R2P2 does not impose any threading model. Given the callback-based design, threads in charge of sending or receiving RPCs operate in a polling loop mode. The library supports symmetric models, where threads are in charge of both network and application processing, by having each thread manage and expose a distinct worker queue through a specific UDP destination port. The DPDK implementation further manages a distinct Tx and Rx queue per thread, and uses Flow Director [36] to steer traffic based on the UDP destination port. In an asymmetric model, a single dispatcher thread links with r2p2-lib, and the other worker threads are in charge of application processing only. This model exposes one worker queue via one UDP destination port.

4.2 Router - software implementation

We implemented a Random, a Round-Robin, a JSQ and a JBSQ(n) policy on the software router. The main implementation requirements for the router are (1) it should add the minimum possible latency overhead, and (2) it should be able to process short REQ0 and R2P2-FEEDBACK messages at line rate. While the router processes only those two types of packets, the order in which it processes them matters. Specifically for JBSQ, the ideal design separates REQ0 from R2P2-FEEDBACK messages into two distinct ingress queues and processes R2P2-FEEDBACKs with higher priority to ensure that the server state information is up-to-date and minimize queuing delays.

Our DPDK implementation uses two different UDP ports, one for each message type, using Flow Director for queue separation. Given the strict priority of control messages and the focus on scalability, we chose a multi-threaded router implementation with split roles for REQ0 threads and R2P2-FEEDBACK threads, with each thread having its own Rx and Tx queues.

JBSQ(n) requires a counter per worker queue that counts the outstanding requests. To minimize cache-coherency traffic, the router maintains two single-writer arrays, one updated on every REQ0 and the other on every R2P2-FEEDBACK, with one entry per worker.

The implementation of the R2P2-FEEDBACK thread is computationally very cheap and embarrassingly scalable. Processing REQ0 messages requires further optimizations to reduce cache-coherency traffic, *e.g.,* maintain the list of known idle workers, cache the current queue sizes, *etc.* Our implementation relies on adaptive bounded batching [9] to amortize the cost of PCIe I/O operations, as well as that of the cache-coherency traffic (the counters are read once per iteration). We limit the batch size to 64 packets.

Finally, we implement a tweak to the JBSQ(n) policy with $n \geq 2$: when no idle workers are present, up to 32 packets are held back for a bounded amount of time on the optimistic view that an R2P2-ACK message may announce the next idle worker. This optimization helps absorb instantaneous congestion and approximate the single-queue semantics in medium load situations.

4.3 P4/Tofino implementation

We built a proof-of-concept P4 implementations of R2P2 router for Tofino [6] using $P4_{14}$ [72]. Similar to the software implementation, the switch only processes REQ0 and

R2P2-FEEDBACK messages and leverages P4 registers to keep soft state. P4 registers are locations in the ASIC SRAM, which can be read and updated from both the control and dataplane.

We focus our description on the implementation of JBSQ(n) for the Tofino dataplane, as the others are trivial in comparison. It consists of 480 lines of P4 source, including header descriptions. Unlike the software implementation that can easily buffer the outstanding REQ0 messages if there is no available server queue, high-performance pipelined architectures, such as Tofino, do not allow buffering in the dataplane. Thus, our P4 logic executes as part of the ingress pipeline of the switch and relies heavily on the ability to recirculate packets through the dataplane via a virtual port. The implementation leverages an additional header that is added to the packet to carry metadata through the various recirculation rounds and is removed before forwarding the packet to the target server.

The logic for REQ0 tries to find a server with $\leq i$ outstanding packets in round i. There is one register instance corresponding to each server, holding the number of outstanding requests. If a suitable server is found, the register value is increased by one, the packet destination is changed to the address of the equivalent server, and the packet is directed to the egress port. We start with $i = 0$ and we increase till $i = n$ from JBSQ(n). When i reaches n and there is still no available server, we keep recirculating the packet without increasing i further. As an optimization to reduce the number of recirculations, the dataplane keeps the i for the last forwarded request and starts from that.

To overcome the Tofino limitation of only being able to compare a limited number of registers in one pass, we also leverage recirculation to inspect the outstanding requests of each bounded queue in each round. Register instances that correspond to different queues are organized in groups that can be checked in one pass. If no available queue is found in the first group, the packet is recirculated (without increasing i) and the second group of queues is checked, *etc.* When a REQ0 arrives, it is initially assigned to a group in a round-robin fashion to further reduce the amount of recirculations.

The logic for R2P2-FEEDBACK decrements the outstanding count for the specific server based on the packet source and consumes the packet without forwarding it.

The use of recirculation has two side-effects: (1) the order of RPCs cannot be guaranteed as one packet may be recirculated while another one is not; (2) the atomicity of the full set of comparisons is not guaranteed as R2P2-FEEDACK packet may be processed while an REQ0 packet is being recirculated. Non-optimal decisions may occur as the result of this race condition.

5 Evaluation

To evaluate the performance and the efficacy of the R2P2 protocol, the two implementations of the router, as well as the trade-offs in using JBSQ(n) over other routing policies, we run a series of synthetic microbenchmarks and two real applications in a distributed setup with multiple servers. The microbenchmarks depend on an RPC service with configurable service time and response size. All our experiments are open-loop [83] and clients generate requests with a Poisson inter-arrival time. We use two baselines and compare them against different configurations for R2P2 with and without the router: (1) vanilla NGINX [66] serving as reverse proxy for HTTP requests; and (2) ZygOS [76], a state-of-the-art work-conserving multicore scheduler. As a load generator we use an early version of Lancet [46].

Our experimental setup consists of cluster of 17 machines connected by a Quanta/Cumulus 48x10GbE switch with a Broadcom Trident+ ASIC. The machines are a mix of Xeon E5-2637 @ 3.5 GHz with 8 cores (16 hyperthreads), and Xeon E5-2650 @ 2.6 GHz with 16 cores (32 hyperthreads). All machines are configured with Intel x520 10GbE NICs (82599EB chipset). To reduce latency and jitter, we configured the machine that measures latency to direct all UDP packets to the same NIC queue via Flow Director. The Barefoot Tofino ASIC runs within a Edgecore Wedge100BF-32X. The Edgecore is directly connected to the Quanta switch via a 40Gbps link and therefore operates as a 1-port router.

5.1 Router characterization

We use the synthetic RPC service to evaluate the latency overhead of the router, the maximal throughput and the optimal request load balancing policy. We configure a setup of 4 servers with 16 threads (64 independent workers), running the synthetic RPC service over DPDK.

Throughput: We first evaluate the sustainable throughput of the software router. We run a synthetic RPC service with 8-byte requests and we configure the size of the response.

Figure 4 shows the achieved goodput as a function of the response size, and compares a configuration with R2P2 messages handled by a JBSQ load balancing policy, with a NGINX configured as reverse proxy for HTTP messages. For small response sizes, the router is bottlenecked by the router's NIC's packets per second (PPS), or the number of outstanding requests in each queue, n in $JBSQ(n)$. JBSQ(3) was enough to achieve maximum throughput. As the response size increases though, the application goodput converges to $4 \times$ 10GbE, the NIC bottleneck of the 4 servers with payloads as small as 2048. Obviously, this is made possible by the protocol itself, which bypasses the router for all REPLY messages. Note that because R2P2 leverages both Direct Server Return and Direct Client Request, even in cases of large requests the router would not be the bottleneck, unlike traditional L4 DSR-enabled load balancing. In contrast, the NGINX I/O bottleneck limits goodput to the load balancer's 10Gbps NIC.

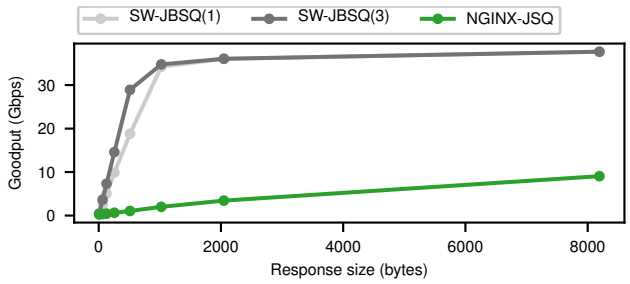


Figure 4: Achieved Goodput as a function of the response size for the JBSQ policy on the software router managing 4 servers connected with 10GbE NICs compared to NGINX configured as HTTP reverse proxy loadbalancing the same 4 servers using a JSQ policy.

Latency overheads and saturation: Figure 5 uses a zero-cost ("echo") RPC service with 8-byte requests and responses, to measure the 99th percentile tail latency as a function of the load for the software middlebox and the Tofino router with the JBSQ policy. As a baseline, we use a DIRECT configuration where clients bypass the router and send requests directly to the servers after a random choice. The figure shows that the latency added by the router is 5µs for the software middlebox and 1µs for the Tofino solution. The software latency is consistent with the characteristics of one-way forwarding performance of the Intel x520 chipset using DPDK. The hardware latency is consistent with the behavior of an ASIC solution that processes and rewrites packet headers in the dataplane. Figure 5 also shows the point of saturation, which corresponds to 7 MRPS for the software middlebox. Given that for every request forwarded the router receives one R2P2-FEEDBACK message, the router handles more than 14M PPS, which is the hardware limit. We were unable to characterize the maximal per-port capability of the Tofino ASIC running the R2P2 logic beyond >8 MRPPS with tiny requests and replies, simply for lack of available client machines. We also observe that the hardware implementation, as expected, requires a smaller n for JBSQ(n). In the figure we show the smallest value of n that achieved maximum throughput.

Comparison of scheduling policies: Figure 6 uses a synthetic $\bar{S} = 25\mu s$ workload to evaluate the different request load balancing policies, implemented on the software router. We evaluate the following policies: DIRECT, where clients bypass the router by making a random server selection, RANDOM where clients talk to the router and the router makes a random selection among the servers, RR where the router selects a target server in a round-robin manner, SW-JBSQ(n) which is the software implementation for the bounded shortest queue with n outstanding requests, and JSQ which is the R2P2 router's implementation of the join-shortest-queue policy. We also compare R2P2 with using NGINX as an HTTP reverse proxy implementing a JSQ policy, which is a vanilla, widely-used

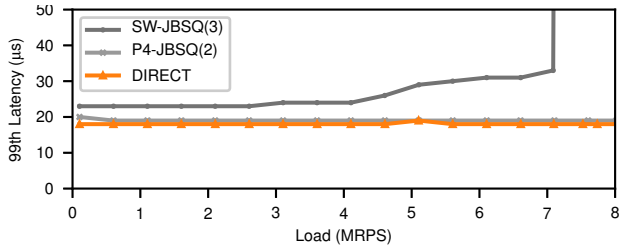


Figure 5: Tail-latency as a function of throughput for a zero service time synthetic RPC service for the software (SW-JBSQ) and the Tofino (P4-JBSQ) implementation of JBSQ compared to DIRECT. In DIRECT clients bypass the router and talk directly to servers by making a random server choice.

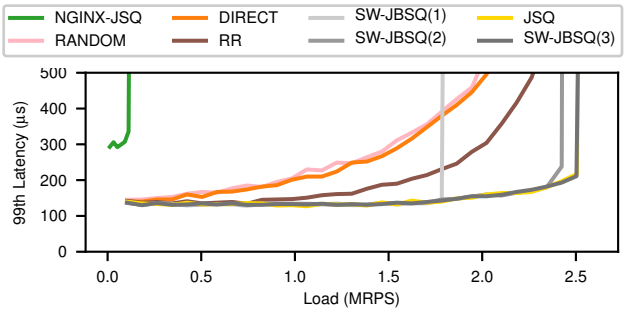


Figure 6: Evaluation of different load balancing policies for an exponential service time workload with $\bar{S} = 25\mu s$.

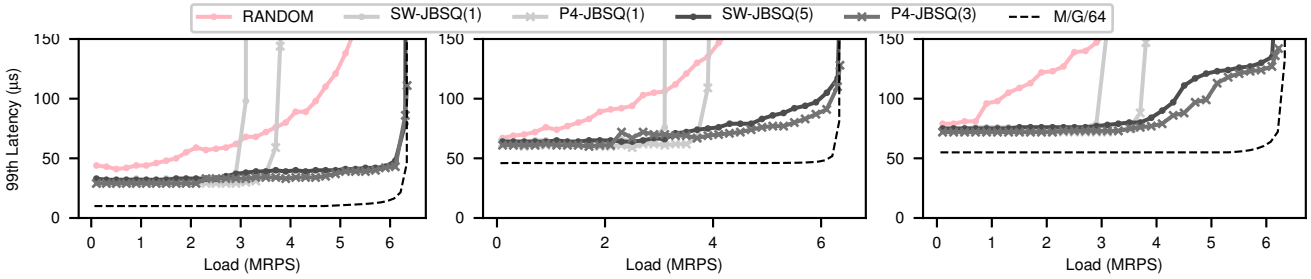
deployment for request-level load balancing.

We make the following observations: (i) NGINX overheads prevent throughput scalability; (ii) DIRECT and RAND configurations perform similarly for R2P2, which is the result of a random choice (in the client or the router equivalently); (iii) RR performs better than random choice, but worse than JBSQ, given the service time dispersion; (iv) JBSQ($n \geq 3$) achieves maximum throughput. Given that the communication time between the server and the router is $\sim 15\mu s$ and the exponential service time dispersion, this is on par with our analysis in § 3.3. (v) JSQ performs similarly to JBSQ(3) for this service time.

5.2 Synthetic Time Microbenchmarks

Figure 7 evaluates JBSQ(n) performance with an aggressive $\bar{S} = 10\mu s$ mean service time and three different service time distributions: Fixed, Exponential and Bimodal where 10% of the request are 10x slower than the rest [55]. We present results for both the software and Tofino implementation, for JBSQ(1) and the optimal n choice for each configuration. Requests and the responses are 8 bytes. We observe:

- For all experiments, all JBSQ(n) variants approximate the optimal single-queue approach ($M/G/64$) until the saturation point for JBSQ(1).



(a) Fixed (64 workers) (b) Exponential (64 workers) (c) Bimodal (64 workers)

Figure 7: Synthetic Service Time Microbenchmarks. Service time $\bar{S} = 10\mu s$.

- Beyond the saturation point of `JBSQ(1)`, an increase in the tail latency as the system configuration trades off higher throughput (*i.e.,* `JBSQ(`$n > 1$`)`) against the use of a theoretically-optimal approach.

- A comparison between the software and hardware implementation shows that more outstanding requests are required for the software implementation; this is because the communication latency between the server and the hardware router is $\sim$5µs faster.

- `JBSQ` achieves the optimal performance, as predicted by the `M/G/64` model, both for the software and the hardware implementation within the 150µs SLO.

- Reducing `n` can have a considerable impact on tail-latency especially in cases with high service time dispersion, as it can be seen in Figure 7c (`SW-JBSQ(5)` vs. `P4-JBSQ(3)`)

5.3 Multi-packet Requests Microbenchmark

R2P2 implements the following logic in splitting requests to packets. If the request fits in a single packet, the whole request payload is transferred with `REQ0`. In the case of a multi-packet request, `REQ0` is a 64-byte packet, carrying only the first part of the request and the rest of the payload is transferred with the `REQN` packets directly to the server. This way the router does not become a throughput bottleneck in the case of large requests, while the extra round-trip is avoided in the case of small requests.

To evaluate the extra round-trip that R2P2 introduces in the case of multi-packet requests with the distinction between `REQ0` and `REQN`, we ran a synthetic microbenchmark with larger requests. Based on the above logic, a 1464-byte request is the biggest request that fits in a single packet given the size of protocol headers. Equivalently, a 1465-byte request is the smallest request that requires 2 packets, and consequently an extra round-trip. We run the synthetic service time RPC server with the bimodal service time distribution of $\bar{S} = 10$ and the 2

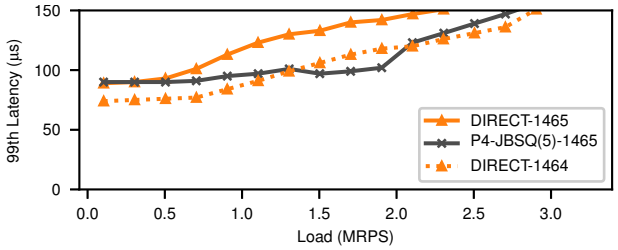


Figure 8: Bimodal service time with $\bar{S} = 10\mu s$ and 64 workers with single and multi-packet requests. `DIRECT-1464` corresponds to an 1-packet request, while `DIRECT-1465` and `P4-JBSQ(5)` correspond to 2-packet requests.

different request sizes. We compare the `DIRECT` deployment with one using the router with the `JBSQ` policy.

Figure 8 summarizes the result of the experiment. We observe that there is a fixed gap of around 15µs between `DIRECT-1464` and `DIRECT-1465` curves that corresponds to the extra round-trip between the client and the server. We, also, run the multi-packet request scenario while using the P4 router with the `JBSQ` policy. We show that despite the extra round-trip, the intermediate hop, and the increased number of packets to process, the 99th percentile latency is close to the single-packet scenario in the `DIRECT` case, which justifies our design decision to pay an extra round-trip to achieve better scheduling.

5.4 Using R2P2 for server work conservation

We now demonstrate how the use of network-based load balancing, *e.g.,* using R2P2, can increase the efficiency of a *single* server scheduling tasks. For this, we compare R2P2 with `JBSQ` with the performance of ZygOS [76], a state-of-the-art system optimized for µs-scale, multicore computing that includes a work-conserving scheduler within a specialized operating system. ZygOS relies on work-stealing across idle cores and makes heavy use of inter-processor interrupts. Both ZygOS and `JBSQ(n)` offer a work-conserving solution to dispatch requests across the multiple cores of a server: Zy-

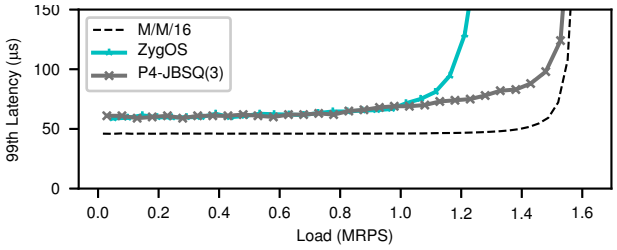


Figure 9: Comparison of R2P2 with the ZygOS [76] work-conserving scheduler: Exponential workload with $\bar{S} = 10\mu s$.

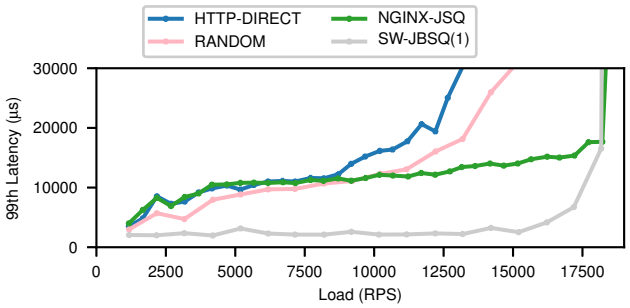


Figure 10: Lucene++ running on 16 16-threaded workers

gOS does it within the server in a protocol-agnostic manner, whereas R2P2 implements the policy in the network.

Figure 9 compares ZygOS with the Tofino implementation of `JBSQ(3)` for the 10µs exponentially-distributed service time workload using a single Xeon server. As in the previous configurations, for the R2P2 implementation each of the 16 Xeon cores, is exposed as a worker with a distinct queue to the router, listening to a different UDP port. In this experiment, the theoretical lower bound is therefore determined by $M/M/16$. We observe that `JBSQ(3)` exceeds the throughput performance of ZygOS, with no visible impact on tail latency despite the additional hop and that `JBSQ(3)` is sufficient to achieves the maximum throughput. For a service-level objective set at 150µs, R2P2 with `JBSQ(3)` outperforms ZygOS by 1.26×. The explanation is that the R2P2 server operates on a set of cores in parallel without synchronization or cache misses, whereas ZygOS has higher overheads due to protocol processing, boundary crossings, task stealing, and inter-processor interrupts.

5.5 Lucene++

Web search is a replicated, read-only workload with variability in the service time coming from the different query types, thus it is an ideal use-case for `R2P2-JBSQ`. For our experiments we used Lucene++ [56], which is a search library ported to serve queries via either HTTP or R2P2. A single I/O thread dispatches one request at a time to 16 Lucene++ worker threads, each of them searching part of the dataset. The experimental setup relies on 16 disjoint indices created from the English Wikipedia page articles dump [91], yielding an aggregated index size of 3.5MB. All indices are loaded in memory at the beginning of the execution to avoid disk accesses. The experimental workload is a subset of the Lucene nightly regression query list, with 10K queries that comprise of simple term, Boolean combinations of terms, proximity, and wildcard queries [57]. The median query service time is 750µs, with short requests taking less than 450µs and long ones over 10ms.

Figure 10 summarizes the experiment results for running Lucene++ on a 16-server cluster, each using 16 threads. The `NGINX-JSQ` and `HTTP-DIRECT` experiments rely on 1568 persistent TCP client connections. First, we observe that `HTTP-DIRECT` over TCP and `RANDOM` over R2P2 which are multi-queue models, have higher tail-latency. Then, we see that `NGINX-JSQ` and `SW-JBSQ(1)` on R2P2 deliver the same throughput; system and network protocol overheads are irrelevant for such coarse-grain workload. Also, $n = 1$ is enough to get maximum throughput, given the longer average service time. `SW-JBSQ(1)` delivers that throughput via the optimal single-queue implementation, with a significant impact on tail latency. As a result, R2P2 lowers the 99^{th} percentile latency by 5.7× at 50% system load over `nginx`.

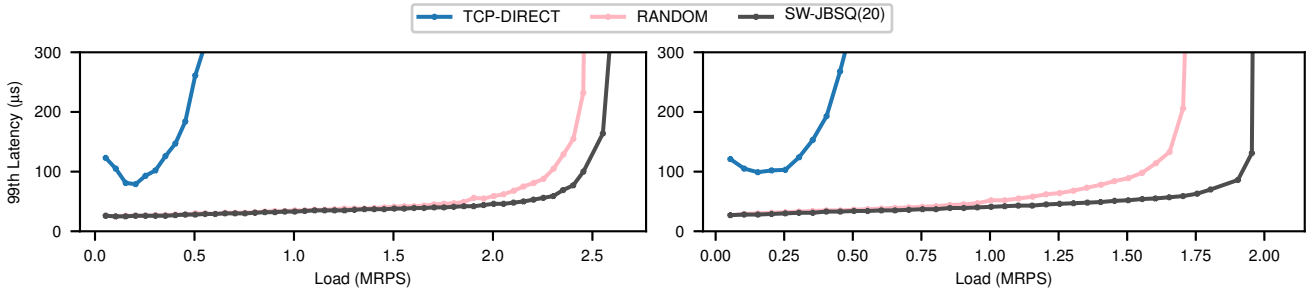
5.6 Redis

Redis [78] supports a master/slave replication scheme with read-only slaves. We ported Redis on R2P2 and ran it on DPDK for the Facebook USR workload [5]. We used the `sticky` R2P2 policy (see §3) to direct writes to the master node and we load balance reads across the master and slave nodes, based on the `RANDOM` and the `JBSQ` policy. Redis has sub-µs service times. Thus, to achieve maximum throughput we had to increase the number of tokens to 20 per worker (`SW-JBSQ(20)`), for the software router. For the vanilla Redis over TCP clients randomly select one of the servers for read requests, while they only send write requests to the master.

Figure 11a shows that R2P2, for an SLO of 200µs at the 99^{th} percentile, achieves 5.30× better throughput for the USR workload over vanilla Redis over TCP (`TCP-DIRECT`) because of reduced protocol and system overheads, while `SW-JBSQ(20)` achieves slightly better throughput than `RANDOM` for the same SLO. Figure 11b increases the write percentage of the workload from 0.2% to 2%, which increases service time variability: R2P2 `RANDOM` has 4.09× better throughput than `TCP-DIRECT`. `SW-JBSQ(20)` further improves throughput by 18%, for a total speedup of 4.8×, as a result of better load balancing decisions.

6 Related work

RPCs can be transported by different IP-based protocols including HTTP2 [10], QUIC [48], SCTP [84], DCCP [47],



(a) Standard USR workload [5] (0.2% writes)

(b) Modified USR with 2% writes

Figure 11: 99^{th} percentile latency vs. throughput for Redis in a 4-node master/slave configuration.

or similar research approaches [4, 28, 30, 32, 63] that identify the TCP limitations and optimize for flow-completion time. Libraries such as gRPC [31] and Thrift [86] abstract away the underlying transport stream into request-reply pairs. Approaches such as eRPC [41] aim at end-host system optimizations and are orthogonal to R2P2. Load balancers proxy RPC protocols such as HTTP in software [23, 66, 69] or in hardware [1, 16, 25, 60]. R2P2 exposes the RPC abstraction to the network to achieve better RPC scheduling, and to the application to hide the complexity of the underlying transport.

Load dispatching, direct or through load balancers, typically *pushes* requests to workers, requiring tail-mitigation techniques [17, 33]. In Join-Idle-Queue [55], workers *pull* requests whenever they are idle. R2P2 additionally supports `JBSQ(n)`, which exposes the tradeoff between maximal throughput and minimal tail latency explicitly.

Task scheduling in distributed big data systems is largely aimed at taming tail-latency and sometimes depends on split-queue designs [19, 20, 44, 71, 75, 77, 92], typically operating with millisecond-scale or larger tasks. R2P2 provides the foundation for scheduling of µs-scale tasks.

Multi-core servers are themselves distributed systems with scheduling and load balancing requirements. This is done by distributing flows using NIC mechanisms [79] in combination with operating systems [24, 73] or dataplane [9, 37, 74] support. Zygos [76] and Shinjuku [40] are an intra-server, work-conserving schedulers for short tasks that rely on task stealing and inter-processor interrupts. R2P2 eliminates the need for complex task stealing strategies by centralizing the logic in the router.

Recent work has focused on key-value stores [54, 59, 70, 78]. MICA provide concurrent-read/exclusive-access (CREW) within a server [54] by offloading the routing decisions to the client, while hardware and software middleboxes [39, 53, 67] or SDN switches [13, 15] enhance the performance and functionality of key-value stores in-network. RackOut extended the notion of CREW to rack-scale systems [68]. R2P2 supports general-purpose RPCs not limited to key-value stores, together with a mechanisms for steering policies which can be used to implement CREW both within a single server and across the datacenter.

Finally, R2P2 adheres and encourages the in-network compute research path by increasing the network visibility to application logic and implementing in-network scheduling. Approaches leveraging in-network compute include caching [39, 53], replicated storage [38], network sequencing [51, 52], DNN training [81, 82], and database acceleration [50].

7 Conclusion

We revisit the requirements to support µs-scale RPCs across tiers of web-scale applications and propose to solve the problem in the network by making RPCs true first-class citizens of the datacenter. We design, implement and evaluate a proof-of-concept transport protocol developed specifically for µs-scale RPCs that exposes the RPC abstraction to the network and at the endpoints. We showcase the benefits of the new design by implementing efficient, tail-tolerant µs-scale RPC load-balancing based on a software router or a programmable P4 ASIC. Our approach outperforms standard load balancing proxies by an order of magnitude in throughput and latency, achieves close to the theoretical optimal behavior for 10µs tasks, reduces the tail latency of websearch by 5.7× at 50% load, and increases the scalability of Redis in a master-slave configuration by more than 4.8×.

Acknowledgements

We would like to thank Katerina Argyraki, Jim Larus, the anonymous reviewers, and our shepherd Mahesh Balakrishnan on providing valuable feedback on the paper. Also, we would like to thank Irene Zhang, Dan Ports and Jacob Nelson for their insights on R2P2. This work was funded in part by a VMWare grant and by the Microsoft Swiss Joint Research Centre. Marios Kogias is supported in part by an IBM PhD Fellowship.

References

[1] A10 Networks. https://www.a10networks.com/.

[2] Mohammad Alizadeh, Albert G. Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data center TCP (DCTCP). In *Proceedings of the ACM SIGCOMM 2010 Conference*, pages 63–74, 2010.

[3] Mohammad Alizadeh, Abdul Kabbani, Tom Edsall, Balaji Prabhakar, Amin Vahdat, and Masato Yasuda. Less Is More: Trading a Little Bandwidth for Ultra-Low Latency in the Data Center. In *Proceedings of the 9th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 253–266, 2012.

[4] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. pFabric: minimal near-optimal datacenter transport. In *Proceedings of the ACM SIGCOMM 2013 Conference*, pages 435–446, 2013.

[5] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of the 2012 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 53–64, 2012.

[6] Barefoot Networks. Tofino product brief. https://barefootnetworks.com/products/brief-tofino/, 2018.

[7] Luiz André Barroso, Jimmy Clidaras, and Urs Hölzle. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines, Second Edition*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2013.

[8] Luiz André Barroso, Jeffrey Dean, and Urs Hölzle. Web Search for a Planet: The Google Cluster Architecture. *IEEE Micro*, 23(2):22–28, 2003.

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

[10] M. Belshe, R. Peon, and M. Thomson. Hypertext Transfer Protocol Version 2 (HTTP/2). RFC 7540 (Proposed Standard), May 2015.

[11] Andrew Birrell and Bruce Jay Nelson. Implementing Remote Procedure Calls. *ACM Trans. Comput. Syst.*, 2(1):39–59, 1984.

[12] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: programming protocol-independent packet processors. *Computer Communication Review*, 44(3):87–95, 2014.

[13] Anat Bremler-Barr, David Hay, Idan Moyal, and Liron Schiff. Load balancing memcached traffic using software defined networking. In *Proceedings of the 2017 IFIP Networking Conference*, pages 1–9, 2017.

[14] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry C. Li, Mark Marchukov, Dmitri Petrov, Lovro Puzar, Yee Jiun Song, and Venkateshwaran Venkataramani. TAO: Facebook's Distributed Data Store for the Social Graph. In *Proceedings of the 2013 USENIX Annual Technical Conference (ATC)*, pages 49–60, 2013.

[15] Eyal Cidon, Sean Choi, Sachin Katti, and Nick McKeown. AppSwitch: Application-layer Load Balancing within a Software Switch. In *Proceedings of the 1st Asia-Pacific Workshop on Networking (APNet)*, pages 64–70, 2017.

[16] Citrix Netscaler ADC. https://www.citrix.com/products/netscaler-adc/.

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

[18] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP)*, pages 205–220, 2007.

[19] Pamela Delgado, Diego Didona, Florin Dinu, and Willy Zwaenepoel. Job-aware Scheduling in Eagle: Divide and Stick to Your Probes. In *Proceedings of the 2016 ACM Symposium on Cloud Computing (SOCC)*, pages 497–509, 2016.

[20] Pamela Delgado, Florin Dinu, Anne-Marie Kermarrec, and Willy Zwaenepoel. Hawk: Hybrid Datacenter Scheduling. In *Proceedings of the 2015 USENIX Annual Technical Conference (ATC)*, pages 499–510, 2015.

[21] Data plane development kit. http://www.dpdk.org/.

[22] Aleksandar Dragojevic, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. FaRM: Fast Remote Memory. In *Proceedings of the 11th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 401–414, 2014.

[23] Daniel E. Eisenbud, Cheng Yi, Carlo Contavalli, Cody Smith, Roman Kononov, Eric Mann-Hielscher, Ardas Cilingiroglu, Bin Cheyney, Wentao Shang, and Jinnah Dylan Hosein. Maglev: A Fast and Reliable Software Network Load Balancer. In *Proceedings of the 13th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 523–535, 2016.

[24] Epollexclusive kernel patch. https://lwn.net/Articles/667087/, 2015.

[25] F5 Networks, INC. https://f5.com/.

[26] Franz Färber, Sang Kyun Cha, Jürgen Primsch, Christof Bornhövd, Stefan Sigg, and Wolfgang Lehner. SAP HANA database: data management for modern business applications. *SIGMOD Record*, 40(4):45–51, 2011.

[27] Fibre channel protocol. https://en.wikipedia.org/wiki/Fibre_Channel_Protocol.

[28] Bryan Ford. Structured streams: a new transport abstraction. In *Proceedings of the ACM SIGCOMM 2007 Conference*, pages 361–372, 2007.

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

[30] Peter Xiang Gao, Akshay Narayan, Gautam Kumar, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. pHost: distributed near-optimal datacenter transport over commodity network fabric. In *Proceedings of the 2015 ACM Conference on Emerging Networking Experiments and Technology (CoNEXT)*, pages 1:1–1:12, 2015.

[31] gRPC. http://www.grpc.io/.

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

[33] Mingzhe Hao, Huaicheng Li, Michael Hao Tong, Chrisma Pakha, Riza O. Suminto, Cesar A. Stuardo, Andrew A. Chien, and Haryadi S. Gunawi. MittOS: Supporting Millisecond Tail Tolerance with Fast Rejecting SLO-Aware OS Interface. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, pages 168–183, 2017.

[34] HAProxy DSR. https://www.haproxy.com/blog/layer-4-load-balancing-direct-server-return-mode/.

[35] Md. E. Haque, Yong Hun Eom, Yuxiong He, Sameh Elnikety, Ricardo Bianchini, and Kathryn S. McKinley. Few-to-Many: Incremental Parallelism for Reducing Tail Latency in Interactive Services. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XX)*, pages 161–175, 2015.

[36] Intel Corp. Intel 82599 10 GbE Controller Datasheet. http://www.intel.com/content/dam/www/public/us/en/documents/datasheets/82599-10-gbe-controller-datasheet.pdf.

[37] Muhammad Asim Jamshed, YoungGyoun Moon, Donghwi Kim, Dongsu Han, and KyoungSoo Park. mOS: A Reusable Networking Stack for Flow Monitoring Middleboxes. In *Proceedings of the 14th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 113–129, 2017.

[38] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. NetChain: Scale-Free Sub-RTT Coordination. In *Proceedings of the 15th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 35–49, 2018.

[39] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. NetCache: Balancing Key-Value Stores with Fast In-Network Caching. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, pages 121–136, 2017.

[40] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. Shinjuku: Preemptive scheduling for µs-scale tail latency. In *Proceedings of the 16th Symposium on Networked Systems Design and Implementation (NSDI)*, 2019.

[41] Anuj Kalia, Michael Kaminsky, and David Andersen. Datacenter RPCs can be General and Fast. In *Proceedings of the 16th Symposium on Networked Systems Design and Implementation (NSDI)*, 2019.

[42] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Using RDMA efficiently for key-value services. In *Proceedings of the ACM SIGCOMM 2014 Conference*, pages 295–306, 2014.

[43] Anuj Kalia, Michael Kaminsky, and David G. Andersen. FaSST: Fast, Scalable and Simple Distributed Transactions with Two-Sided (RDMA) Datagram RPCs. In *Proceedings of the 12th Symposium on Operating System Design and Implementation (OSDI)*, pages 185–201, 2016.

[44] Konstantinos Karanasos, Sriram Rao, Carlo Curino, Chris Douglas, Kishore Chaliparambil, Giovanni Matteo Fumarola, Solom Heddaya, Raghu Ramakrishnan, and Sarvesh Sakalanaga. Mercury: Hybrid Centralized and Distributed Scheduling in Large Shared Clusters. In *Proceedings of the 2015 USENIX Annual Technical Conference (ATC)*, pages 485–497, 2015.

[45] Marios Kogias and Edouard Bugnion. Flow Control for Latency-Critical RPCs. In *Proceedings of the 2018 SIGCOMM Workshop on Kernel Bypassing Networks*, KBNets'18, pages 15–21. ACM, 2018.

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

[47] E. Kohler, M. Handley, and S. Floyd. Datagram Congestion Control Protocol (DCCP). RFC 4340 (Proposed Standard), March 2006. Updated by RFCs 5595, 5596, 6335, 6773.

[48] 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. The QUIC Transport Protocol: Design and Internet-Scale Deployment. In *Proceedings of the ACM SIGCOMM 2017 Conference*, pages 183–196, 2017.

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

[50] Alberto Lerner, Rana Hussein, and Philippe Cudré-Mauroux. The Case for Network Accelerated Query Processing. In *Proceedings of the 9th Biennial Conference on Innovative Data Systems Research (CIDR)*, 2019.

[51] Jialin Li, Ellis Michael, and Dan R. K. Ports. Eris: Coordination-Free Consistent Transactions Using In-Network Concurrency Control. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, pages 104–120, 2017.

[52] Jialin Li, Ellis Michael, Naveen Kr. Sharma, Adriana Szekeres, and Dan R. K. Ports. Just Say NO to Paxos Overhead: Replacing Consensus with Network Ordering. In *Proceedings of the 12th Symposium on Operating System Design and Implementation (OSDI)*, pages 467–483, 2016.

[53] Xiaozhou Li, Raghav Sethi, Michael Kaminsky, David G. Andersen, and Michael J. Freedman. Be Fast, Cheap and in Control with SwitchKV. In *Proceedings of the 13th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 31–44, 2016.

[54] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. MICA: A Holistic Approach to Fast In-Memory Key-Value Storage. In *Proceedings of the 11th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 429–444, 2014.

[55] Yi Lu, Qiaomin Xie, Gabriel Kliot, Alan Geller, James R. Larus, and Albert G. Greenberg. Join-Idle-Queue: A novel load balancing algorithm for dynamically scalable web services. *Perform. Eval.*, 68(11):1056–1071, 2011.

[56] Lucene++. https://github.com/luceneplusplus/LucenePlusPlus.

[57] Lucene nightly benchmarks. https://home.apache.org/~mikemccand/lucenebench.

[58] David Meisner, Christopher M. Sadler, Luiz André Barroso, Wolf-Dietrich Weber, and Thomas F. Wenisch. Power management of online data-intensive services. In *Proceedings of the 38th International Symposium on Computer Architecture (ISCA)*, pages 319–330, 2011.

[59] Memcached. https://memcached.org/.

[60] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. SilkRoad: Making Stateful Layer-4 Load Balancing Fast and Cheap Using Switching ASICs. In *Proceedings of the ACM SIGCOMM 2017 Conference*, pages 15–28, 2017.

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

[62] In-memory mongodb. https://docs.mongodb.com/manual/core/inmemory/.

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

[64] Nginx. https://www.nginx.com/.

[65] NGINX DSR: IP Transparency and Direct Server Return with NGINX and NGINX Plus as Transparent Proxy. https://www.nginx.com/blog/.

[66] NGINX Reverse Proxy. https://docs.nginx.com/nginx/admin-guide/web-server/reverse-proxy/.

[67] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. Scaling Memcache at Facebook. In *Proceedings of the 10th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 385–398, 2013.

[68] Stanko Novakovic, Alexandros Daglis, Dmitrii Ustiugov, Edouard Bugnion, Babak Falsafi, and Boris Grot. Mitigating Load Imbalance in Distributed Data Serving with Rack-Scale Memory Pooling. *ACM Trans. Comput. Syst.*, 36(2):6:1–6:37, 2019.

[69] Vladimir Andrei Olteanu, Alexandru Agache, Andrei Voinescu, and Costin Raiciu. Stateless Datacenter Loadbalancing with Beamer. In *Proceedings of the 15th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 125–139, 2018.

[70] 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. The RAMCloud Storage System. *ACM Trans. Comput. Syst.*, 33(3):7:1–7:55, 2015.

[71] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. Sparrow: distributed, low latency scheduling. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, pages 69–84, 2013.

[72] The P4 Language Specification. https://p4.org/p4-spec/p4-14/v1.0.4/tex/p4.pdf. Accessed on 20.09.2018.

[73] Aleksey Pesterev, Jacob Strauss, Nickolai Zeldovich, and Robert Tappan Morris. Improving network connection locality on multicore systems. In *Proceedings of the 2012 EuroSys Conference*, pages 337–350, 2012.

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

[75] Russell Power and Jinyang Li. Piccolo: Building Fast, Distributed Programs with Partitioned Tables. In *Proceedings of the 9th Symposium on Operating System Design and Implementation (OSDI)*, pages 293–306, 2010.

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

[77] Jeff Rasley, Konstantinos Karanasos, Srikanth Kandula, Rodrigo Fonseca, Milan Vojnovic, and Sriram Rao. Efficient queue management for cluster scheduling. In *Proceedings of the 2016 EuroSys Conference*, pages 36:1–36:15, 2016.

[78] Redis. https://redis.io/.

[79] Microsoft corp. receive side scaling. http://msdn.microsoft.com/library/windows/hardware/ff556942.aspx.

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

[81] Amedeo Sapio, Ibrahim Abdelaziz, Abdulla Aldilaijan, Marco Canini, and Panos Kalnis. In-Network Computation is a Dumb Idea Whose Time Has Come. In *Proceedings of The 16th ACM Workshop on Hot Topics in Networks (HotNets-XVI)*, pages 150–156, 2017.

[82] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan R. K. Ports, and Peter Richtárik. Scaling Distributed Machine Learning with In-Network Aggregation. *CoRR*, abs/1903.06701, 2019.

[83] Bianca Schroeder, Adam Wierman, and Mor Harchol-Balter. Open Versus Closed: A Cautionary Tale. In *Proceedings of the 3rd Symposium on Networked Systems Design and Implementation (NSDI)*, 2006.

[84] R. Stewart. Stream Control Transmission Protocol. RFC 4960 (Proposed Standard), September 2007. Updated by RFCs 6096, 6335, 7053.

[85] Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. The End of an Architectural Era (It's Time for a Complete Rewrite). In *Proceedings of the 33rd International Conference on Very Large DataBases (VLDB)*, pages 1150–1160, 2007.

[86] Apache thrift. https://thrift.apache.org/.

[87] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, pages 18–32, 2013.

[88] Voltdb. https://www.voltdb.com/.

[89] Xingda Wei, Jiaxin Shi, Yanzhe Chen, Rong Chen, and Haibo Chen. Fast in-memory transaction processing using RDMA and HTM. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*, pages 87–104, 2015.

[90] Adam Wierman and Bert Zwart. Is Tail-Optimal Scheduling Possible? *Operations Research*, 60(5):1249–1257, 2012.

[91] The english wikipedia page article dump. https://dumps.wikimedia.org/enwiki/20180401.

[92] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. Apache Spark: a unified engine for big data processing. *Commun. ACM*, 59(11):56–65, 2016.

[93] Heng Zhang, Mingkai Dong, and Haibo Chen. Efficient and Available In-memory KV-Store with Hybrid Erasure Coding and Replication. In *Proceedings of the 14th USENIX Conference on File and Storage Technologie (FAST)*, pages 167–180, 2016.

[94] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. Congestion Control for Large-Scale RDMA Deployments. In *Proceedings of the ACM SIGCOMM 2015 Conference*, pages 523–536, 2015.