Tail-Latency-Tolerant Load Balancing of Microsecond-scale RPCs

Paper #248

Abstract

We focus on the open problem of taming end-to-end tail latency for microsecond-scale remote procedure calls to replicated, scale-out services such as web-search, object streaming, or multiple-reader key-value stores and NoSQL databases running on datacenter leaf nodes. These leaf nodes are typically called by hundreds of independent aggregators that, in turn, respond to user-facing requests.

We propose R2P2, a simple UDP-based transport protocol designed for RPCs. R2P2 allows efficient and scalable RPC routing as it separates the RPC target selection from request and reply streaming. R2P2 supports a number of load-balancing policies, including a novel *join-bounded-shortest-queue (JBSQ)* policy, which lowers tail latency by centralizing pending RPCs in the router and ensuring 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 can route RPCs at line-rate using only two CPU cores. R2P2 improves the tail latency of web index searching on a cluster of 16 workers operating at 50% of capacity by 6× 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 SLO of 200µs by more than 6.1× vs. vanilla Redis. The improvements in both throughput and tail latency are due to the cumulative benefits of a leaner protocol, kernel bypass, and improved scheduling policies. Finally, we demonstrate that the use of a network-based load-balancer can improve the single-server performance for microsecond-scale tasks over a state-of-the-art work-conserving scheduler that implements task stealing using inter-processor interrupts.

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, 18]. In such deployments, a single application can comprise hundreds of software components, deployed on thousands of servers organized in multiple tiers and interconnected by commodity Ethernet switches. The typical pattern for web-scale applications distributes all critical data (*e.g.,* the social graph) in the memory of hundreds of data services, such as memory-resident transactional databases [27,79,81–83], NoSQL databases [57,74], key-value stores [23, 48, 53, 62, 87], or specialized graph stores [14]. This leads to a high fan-in, high fan-out connection graph between tiers of the application that directly communicate using remote procedure calls (RPC) [11]. Each client must (a) fan-out the RPC to the different shards holding data and (b) within each shard, choose a server among the replica set. Overall, each individual task often requires only a handful of µs of user-level execution time for simple key-value requests [48] to a handful of milliseconds for search applications [38].

Figure 1 illustrates the two extreme approaches regarding RPC service deployment of an online data-intensive (OLDI) application with root servers serving end-user queries and leaf servers holding replicated, sharded data [8, 52]: (a) use direct connections and randomized selection of replicas nodes between these two tiers of servers, leading to high fan-in, high fan-out communication patterns, load-imbalance and head-of-line blocking or (b) use a load-balancer to select among replicas on a per request basis, *e.g.,* using a Round-Robin or Join-Shortest-Queue algorithm (JSQ).

Unlike previous work focused on flow-completion-time targeting the delivery of short RPCs [4,31,33,35,58], we deal with the problem of end-to-end RPC tail-latency, inclusive of the queuing delays and the service time within the RPC server. Our work focuses on the latency tail-tolerant routing of RPCs destined to replicated, fine-grain, in-memory

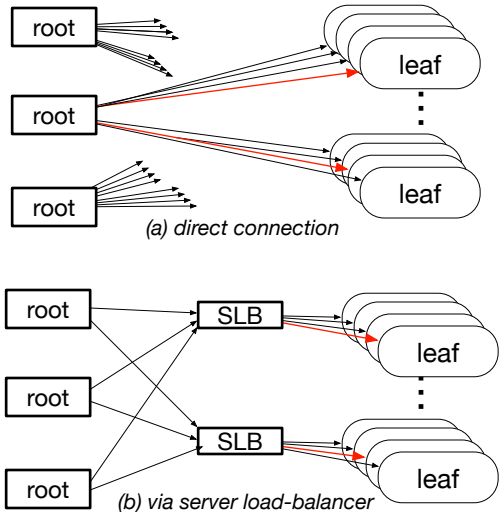


Figure 1: Deployment options in two-tier workloads with multiple root aggregators and replicated shards, resulting in both high fan-in and fan-out.

services. The theoretical background is well known: (a) single-queue, multiple-processor models deliver lower tail latency than parallel single-queue, single-processor models; (b) *First-Come-First-Serve* (FCFS) delivers the best tail latency for low-dispersion tasks while processor sharing delivers superior results in high dispersion service time distributions [84]; (c) load-balancers with a *Join-Shortest-Queue* (JSQ) strategy perform well under medium load for low-dispersion tasks; (d) distributed load balancers such as *Join-Idle-Queue* (JIQ) [49] outperform power-of-choice algorithms [56] at high-loads and lower tail latency by having the servers pull requests from intermediate dispatchers.

The paradigm mismatch between TCP, which is a byte-oriented, streaming transport protocol, and RPCs which are message oriented, has made the practical answers less obvious: centralized load-balancers such as the commercial application delivery controllers from A10, Citrix and F5 [1,17,26] proxy standard protocols such as HTTP but add baseline latency and create scalability and I/O bottlenecks. Further, their classic JSQ strategy leads to queue buildups in servers, which creates tail-imbalance at high load, in particular for high-dispersion service times.

In this work, we propose to redesign the underlying network transport protocol with the explicit goal of ensuring the efficient, scalable, tail-tolerant, high-throughput routing of RPCs. The design includes a request router which can be implemented in an efficient manner either in software or within a programmable switch ASIC such as P4 [13] for different routing policies. The protocol is scalable in the presence of streaming requests and/or replies by separating the datapath from the RPC control path. Router supports classic load-balancing policies. In addition, we propose *Join-Bounded-Shortest-Queue* (`JBSQ(n)`), an RPC scheduling policy that splits queues between the router and the servers, allowing only a bounded number of outstanding requests per server to significantly improve tail-latency.

We make the following contributions :

1. The design of *Request-Response Pair Protocol (R2P2)*, a transport protocol designed for datacenter RPCs that separates request selection from message streaming, nearly eliminates head-of-line blocking and efficiently load balances requests. R2P2 exposes a non-POSIX API to applications with RPC-tailored semantics.

2. The implementation of R2P2 router on a software middlebox that adds only 5µs to the end-to-end unloaded latency and is capable of load balancing incoming RPCs at line rate using only 2 cores.

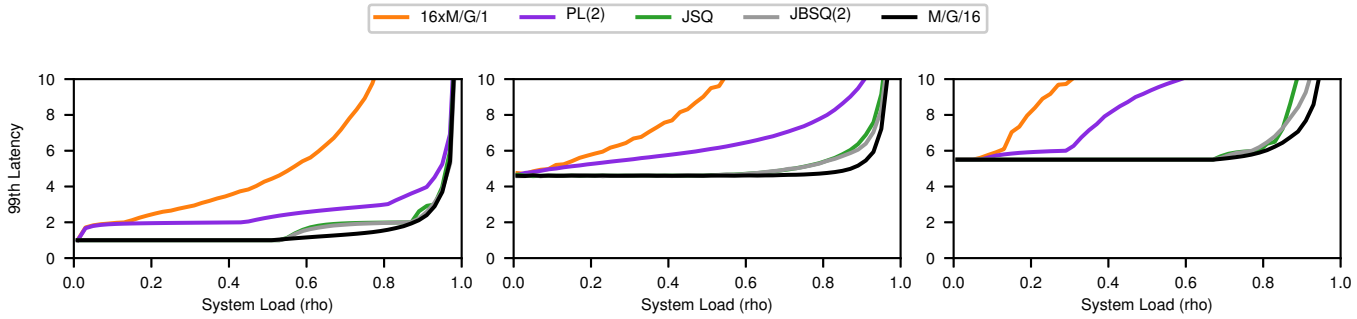
3. The implementation of R2P2 and of the `JBSQ(n)` policy within a P4-programmable Tofino dataplane ASIC, which eliminates the I/O bottlenecks of a software middlebox and reduces latency overhead to 1µs.

4. A queueing-theoretic analysis of RPC scheduling. We show that `JBSQ(n)` performs close to the optimal single-queue model.

Our evaluation with microbenchmarks shows that our R2P2 deployment with a Join-Bounded-Shortest-Queue 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++ [50], 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 6× at 50% system load over `nginx` with 16 workers. Finally, running Redis [74], a popular key-value store with built-in master-slave replication, over R2P2 demonstrates an increase of 5.85×– 6.1× in throughput vs. vanilla Redis (over TCP) at a 200µs tail-latency SLO, 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 and introduces the problem. §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.

All source code will be available in open-source at publication time.



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

Figure 2: Simulation results for the 99^{th} percentile latency across 3 service time distributions with $\overline{S} = 1$

2 Background

2.1 Datacenter RPCs

TCP has emerged as the main transport protocol for latency-sensitive, intra-datacenter RPCs on top of commodity hardware, as its reliable stream semantics provide a convenient abstraction for developers 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 even the replies are small and can fit in a single packet [5, 58].

RPC semantics: Many datacenter applications choose weak consistency models [19] to lower tail latency. Such 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 each other. Thus, the reliable, in-order stream provided by TCP is a lot stronger than what the applications need, that comes with the 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 upper-level protocols support multiple outstanding RPCs on the same flow, as is the case with `http/2`, memcache, *etc.*. Multiplexing RPCs on the same flow implies ordering of the requests that share a socket, despite the individual RPCs being semantically independent. This ordering imposes scheduling limitations and Head-of-Line-Blocking (HOL). HOL appears when small requests are stuck behind a big request, or in case of packet drops.

Connection scalability: RPCs in datacenter applications lead to high fan-out patterns in clients and high fan-in in servers, often beyond the design efficiency point of commodity operating systems. 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` [62].

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 [43]. Applications willing to trade-off harvest vs. yield [32] would ideally never issue RPCs buffered in the network stack with no chance of returning by the deadline.

2.2 Load balancing

The problem of spreading out load extends to load balancing across servers within a distributed, scale-out environment. Load balancers hide a set of servers behind a single virtual address and improve the availability and capacity of applications. Load-balancing decisions can severely affect throughput and tail-latency; thus, a significant amount of infrastructure is dedicated to load balancing [24,54]. Load balancers can be implemented either in software [24, 59, 64] or in hardware [1, 17, 26, 54] and fall into two broad categories. Layer-4 ("network") load balancers use the 5-tuple information of the TCP or UDP flow to select a destination server and the assignment is static and independent of load. Layer-7 ("application") load balancers come in the form of `http` reverse proxies or protocol-specific routers [62]. These load balancers terminate the client TCP connections and, they use dynamic policies to select a destination server and reissue the request to the server on a different connection. There are several policies that can be used to decide the eventual RPC target, including random, power-of-two [56], round-robin, Join-Shortest-Queue (JSQ), and Join-Idle-Queue (JIQ) [49]. While ubiquitous at the web tier, layer-7 load balancers are

less commonly deployed within tiers of applications to support µs-scale RPCs. The reasons for this are that (i) the increased latency associated with the use of the proxy and (ii) the scalability issues introduced when all requests and responses flow through proxies.

2.3 As a Queuing Theory problem

Taming tail latency in networked systems is largely an exercise in identifying, managing, and bounding queues of the distributed system, and subsequently in scheduling the tasks on worker nodes. So, in this section we approach the problem of RPC load balancing from a theoretical point of view by abstracting away the system aspect through basic queueing theory.

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 [46, 84].

Between those two extremes, there are other models that improve upon random selection.Power-of-two [56] (`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 [49].

We define Join-Bounded-Shortest-Queue `JBSQ(n)` as a policy which 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 2 quantifies the tail-latency benefit, at the 99^{th} percentile, for these queueing models for a configuration with a Poisson arrival process, $k = 16$ workers, and three well-known distributions with the same service time $\bar{S} = 1$: deterministic, exponential and bimodal-1 (where 90% of requests execute in .5 and 10% in 5.5 units) [49], provided by discrete event simulation.

We can conclude the following from the simulation results. There is a dramatic gap in performance between the random, multi-queue model, and the optimal single-queue approach.[1] `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. `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 take into account 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, and minimize tail latency and head-of-line blocking even at very high loads.

[1] The single-queue is optimal among FCFS queuing systems. There is no universally optimal scheduling strategy for tail-latency [84].

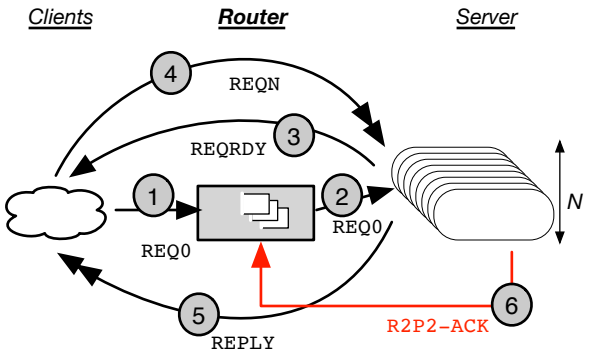


Figure 3: The R2P2 protocol for a request-reply exchange. Each message is carried within a UDP packet. Single arrows represent a single packet whereas double arrows represent a stream of datagrams.

3 R2P2: A Transport Protocol for RPCs

We propose R2P2 (*Request-Response Pair Protocol*), a simple UDP-based transport protocol specifically targeting RPCs within a distributed infrastructure, *e.g.,* datacenters, aiming to improve end-to-end tail latency, through efficient request-level load balancing. Our goal is to implement a `JBSQ(n)` routing policy, which approximates the optimal single-queue model (see §2.3).

R2P2 is a connectionless transport protocol capable of supporting upper-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 requiring no state across requests. The main entity in R2P2 is a request-response pair. A request-response pair is initiated by the client and is uniquely identified by a triplet of $< src\_IP, src\_port, req\_id >$.

Figure 3 describes the interactions and the packets exchanged for sending and receiving an RPC in the general case of a multi-packet request and a multi-packet response, within a distributed infrastructure using a request router to load balance requests across the servers.

1. a `REQ0` message opens the RPC interaction, uniquely defined by the combination of source IP, UDP port, and

a unique 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, according to the policy, there is no available server, requests can temporarily queue up in the router;

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

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

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

6. The servers send `R2P2-ACK` to the router to signal idleness, availability, or health, depending on the load balancing policies. In the case of `JBSQ(n)` one `R2P2-ACK` message signals the completion of one request.

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 source than the destination used, thus responses are sent directly to the client bypassing the router; (ii) there is no head-of-line blocking resulting from multiplexing multiple RPCs on the same socket, since there are no sockets and each request-response pair is treated independently; (iii) there are no ordering guarantees across different pairs. (iv) the protocol is suited for both short and long RPCs; by avoiding the router for subsequent `REQn` message and replies, the router capacity is only limited by the device's hardware packet processing rate, and not the overall amount of bandwidth of the messages.

R2P2 follows the end-to-end argument in systems design [76]: A client application initiates a request-response pair and is fully in-charge of the fate of each RPC according to its specific needs and SLOs. Cases that require the client application intervention include specific protocol error messages indicating failure, *e.g.,* explicit request drop, or timer timeouts potentially indicating packet loss. By propagating failures to the application, the application is free to choose between *at-least-once* and *at-most-once* semantics per-RPC by re-issuing the same request that potentially failed. Failures affect only the RPC in question without impacting other requests (*e.g.,* head-of-line blocking). This is useful in cases with fan-out replicated requests. Unlike protocols that blindly provide reliable message delivery, R2P2 exposes failures and timeouts to the application, thus providing system support for the implementation of tail mitigation techniques, such as hedged requests [18].

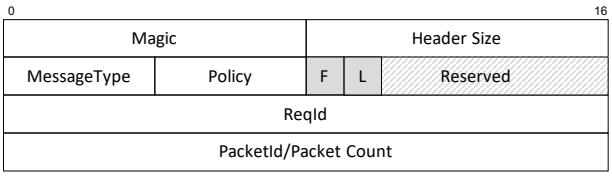


Figure 4: R2P2 Header Format

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

Table 1: The R2P2 message types

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 [28]). For example, a single-packet-request-multi-packet-response RPC over R2P2 would be most 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 4 describes a proof-of-concept R2P2 header, while Table 1 includes the different R2P2 messages. R2P2 supports a 16-bit request id, thus each client (`(src_ip, src_port)` pair) can have up to 65536 outstanding RPCs. Also, the R2P2 header includes a 16-bit packet id meaning that each R2P2 message can consist up to 65536 MTU-sized packets. 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 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 core to our implementation of a tail-tolerant Redis key-value store based on a master/slave architecture; it is used to direct writes to the master, but balances reads according to the load balancing policy.

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. In such environments, packet losses due to transmission errors or packet corruption are exceedingly rare. So, packet loss is the result of congestion and buffer overflows. We make no assumptions about the core network that can depend either on ECMP flow hashing or packet spraying [33, 35, 58]. 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.

Timer management: Given that the assumed deployment model allows for packet reordering, the packet loss detection depends on timers. There are different timers on the server and the client side. 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. There is only one timer on the client side since the retransmission timeouts are comparable with strict latency SLOs. The server maintains one timer per multi-packet request. If part of a multi-packet request is lost and the server does not receive anything for more than `RTO`, it sends a `SACK` message to the client asking for the lost packets. This RTO can be as aggressive as few hundreds of µs equivalent of the propagation delay on the longest path and a congested link [55]. Finally, the server periodically needs to clean-up failed multi-packet requests.

Congestion management: R2P2 focuses on reducing queuing on the server side and is orthogonal to congestion control. Large requests and responses, larger than the bandwidth delay product, will require a congestion management scheme. For such cases, R2P2 is complementary and can leverage receiver-driven packet pacing such as Homa [58] that only paces large message, or ExpressPass [16] if packet spray is not applicable. Alternatively, R2P2 could incorporate schemes based on ECN similar to DCTCP [2] and DCQCN [88] for large messages only. Both approaches will require the addition of a message either to carry the receiver grants or echo the ECN back to the sender.

3.2 Router design considerations

R2P2 was designed having request-level load balancing as a major end-goal. In this section we discuss the design choices regarding the R2P2 request router.

Direct Client Request - Direct Server Return: R2P2 implements Direct Server Return (DSR) [37, 60] since the replies do not go through the router. This is a widely-used technique in L4 load balancers with static policies [60]. R2P2 uses DSR while achieving request-level load balancing. In addition, R2P2 implements Direct Client Request, where the router handles only the first packet of a multi-packet request, and the rest is streamed directly to the corresponding server, thus avoiding the IO bottleneck at the router.

Active Queue Management: Although standard *push* policies can be implemented on the router, `JBSQ(n)` has several benefits in terms of both scheduling and active queue management: (1) `JBSQ(n)` implies buffering of pending requests in the router, leading to extremely shallow queues in the servers, which lowers the tail latency of request processing (see §2.3). (2) The router can easily detect overloaded cases where admission control is warranted, *e.g.,* based on a simple policy to cap the maximum number of outstanding requests based on the known mean service rate and the service-level objective. In such situations, the router can take active queue management decisions, *e.g.,* RED [30], and clients are notified of cancelled RPCs as a flow control mechanism. For example, assume a latency-critical application with tight tail-latency SLO that accepts partial results rather than delayed ones. In such a case, the router can drop RPCs that have already violated their SLO while queued up.

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 `n` values than predicted by the heuristic.

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 can be lost, *e.g.,* the current size of the per-worker bounded queue in the case of `JBSQ(n)`. In that case, the relationships between the router and the servers can be reconstructed by initializing a new set of credits, and the clients can simply failover to using a new virtual IP address (VIP). Additionally, the `REQ0` messages buffered by the router at the time of failure, will be lost. This, though, will be handled by R2P2 as any other `REQ0` loss.

Server membership: Servers behind the R2P2 router can fail or new server can join. `R2P2-ACK` messages implicitly confirm to the router that a server is alive. In case of a failure, the lack of R2P2-ACK messages will prevent the router from sending requests to the failed server. Similarly, newly-added servers can send `R2P2-ACK` messages to the router

informing about their availability to serve requests.

Deployment: The software R2P2 router is deployed as an L4 middlebox. The P4 R2P2 router, though, can be deployed both as a L4 middlebox inside the datacenter fabric or at the top-of-the-rack switch. In the latter case, based on the application requirements and the router implementation the `R2P2-ACK` messages to determine the size of the bounded queues can be reduced, as all traffic goes through the switch, which allows it to know the depth of each queue.

R2P2-direct: Unlike HTTP over TCP, which was designed for a single back-end server and implements request-level load balancing through costly reverse proxying, R2P2 was designed with request-level load balancing in mind, but also supports direct-server access out of the box. Clients can directly send requests to servers without the intermediate load balancing step. Multicore servers can then use well-known mechanisms, *e.g.,* RSS [75] or Flow Director [29], to load-balance the incoming RPCs across different cores.

3.3 API

Trying to eliminate the application code that implements the RPC logic on top of a byte stream abstraction, R2P2 exposes a non-POSIX API specifically designed for RPC workloads. Table 2 summarizes the corresponding application calls and callbacks for the client and server application. The API is independent of the underlying implementation, and 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. The `R2P2-ACK` messages are managed implicitly by the R2P2 implementation on the server. In `JBSQ(n)` the server sends one `R2P2-ACK` message to the router for each RPC reply sent. Although `R2P2-ACK` batching is possible, there is a trade-off between the generated traffic and how up-to-date the router is.

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

4 Implementation

We implemented R2P2 in userspace as a Linux library, `r2p2-lib`, on top of either UDP sockets or DPDK [22] (§4.1). We implemented the software R2P2 router on top of DPDK (§4.2) and the hardware solution in the P4 v1.4 programming language [67] 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 UDP socket and DPDK-based implementations. Our 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 report a distinct worker queue to the router, exposed on a distinct UDP destination port. The DPDK implementation manages a distinct Tx and Rx queue per thread, and uses Flow Director [29] to steer traffic based on the UDP 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 to the router.

4.2 Router - software implementation

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

In our DPDK implementation we use 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-ACK` threads, with each thread having its own Rx and Tx queues.

`JBSQ(n)` requires a counter per worker queue that counts the outstanding requests. Unfortunately, the naive implementation suffers from constant cache-coherency traffic. Instead, the router maintains two single-writer arrays, one updated on every `REQ0` and the other on every `R2P2-ACK`, with one entry per worker.

The implementation of the `R2P2-ACK` thread is computationally very cheap and embarrassingly scalable. The implementation of the `REQ0` thread requires further optimizations to reduce cache-coherency traffic, *e.g.,* maintain the list of known idle workers, cache the subtractions, *etc.* In addition, 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. Note that such optimizations would not be possible in a strict `JSQ` implementation.

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 a feedback 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 Implementation

We built two proof-of-concept P4 implementations of R2P2 router: (1) for a Tofino P4 ASIC [6] using Barefoot's P4 v1.4 [67], and (2) using the p4c-bm2-ss [68] compiler with the BMv2 behavioral model [12] for P4 v1.6 [15]. The difference in language version is due to current restrictions in the compiler for the ASIC.

The R2P2 pipeline only processes `REQ0` and `R2P2_ACK`. All other messages are routed at layer-3. Similar to the software implementation, P4 keeps a count of outstanding requests in a per-worker register. We chose registers for that, as they can be read and updated from both the control and the data plane.

We focus our description on the implementation of `JBSQ(n)` for the Tofino dataplane, as the others are trivial in comparison. It consists of 350 lines of P4 source, including header descriptions. The 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 unused fields in the packet header to carry metadata about the packet through the various recirculation rounds:

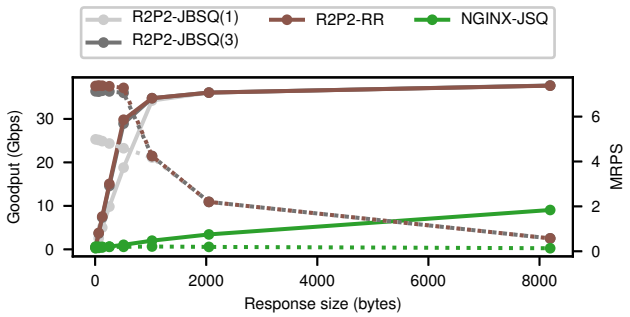


Figure 5: Router Throughput Scalability with 4 Servers (Dashed lines correspond to RPS)

1. The logic for `REQ0` identifies the first ports with $\leq i$ outstanding packets in round i. If a suitable port is found, the register state is updated and the packet is rewritten and directed to the egress port corresponding to the worker.

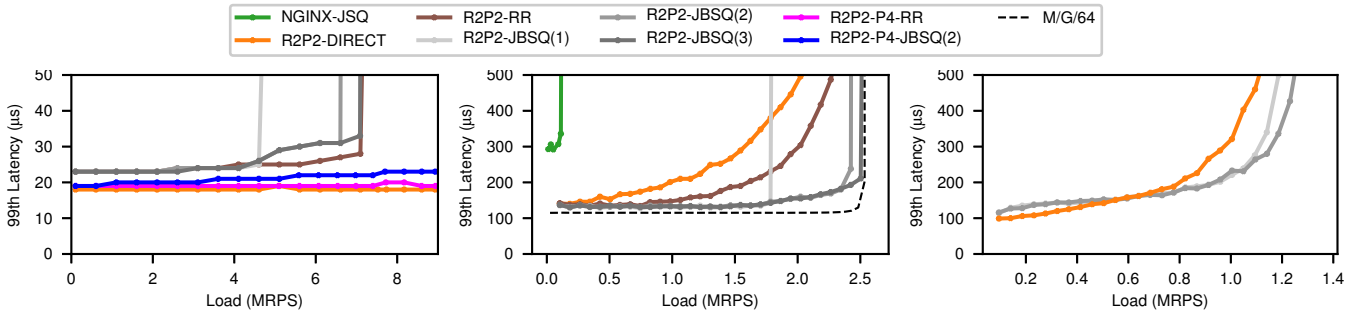
2. As the logic can only compare a limited number of register per pass through the ingress pipeline (an ASIC implementation-specific restriction), packets are also recirculated within a round, and each pass considers only a range of workers.

3. The logic for `R2P2-ACK` decrements the outstanding count and consumes the packet without forwarding it.

The use of recirculation has two side-effects: (1) the order of RPC 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-ACK` packet may be processed while an `REQ0` packet is being recirculated. Non-optimal decisions may occur as the result of this race condition.

The use of registers may also pose an ASIC-specific restriction with respect to the maximal number of workers supported. We were successful in experimenting with up to 64 workers without problems (and did not attempt experiments with a larger sets).

Our implementation of the same protocol on the BMv2 behavioral model in P4 v1.6 is more straighforward: the ability for a pipeline stage to have an unbounded number of register comparisons eliminates the need to break down a round into multiple passes. Packets are only recirculated when all servers have the maximum number of outstanding requests and the packets must be held back. Finally, the atomic annotation of the p4 language allows for the elimination of race conditions.



(a) Zero Service Time - PPS Saturation

(b) Exponential Service Time $\overline{S} = 25\mu s$

(c) Bimodal Response Size $\overline{M} = 3277$ bytes

Figure 6: Router Latency Evaluation

5 Evaluation

To evaluate the performance and the efficacy of R2P2 protocol, the two implementations of the router, as well as the tradeoffs 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 [77] 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 [61] serving as reverse proxy for HTTP requests; and (2) ZygOS [72], a state-of-the-art work-conserving multicore scheduler.

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 a synthetic RPC service to evaluate the latency overhead of the router and the maximal throughput. For this, we compare the load-balanced workload with a variant where each client randomly selects a server for each request, bypassing any router (`R2P2-DIRECT`). We configure a setup of 4 servers with 16 threads (64 independent workers), running the synthetic RPC service over DPDK.

Software middlebox: We first evaluate the DPDK router. Our goal is to identify the sustainable throughput, the added latency overhead, and the optimal routing policy.

Figure 5 shows both the goodput in Gbps and the requests per second as a function of the response size of synthetic requests. We observe that, for small response sizes, the router is bottlenecked by the router's NIC's packets per second (PPS), which limits protocol processing to ~ 7 MRPS. Given that for every request forwarded the router receives one control message, the router handles more than 14M PPS, which is the hardware limit.

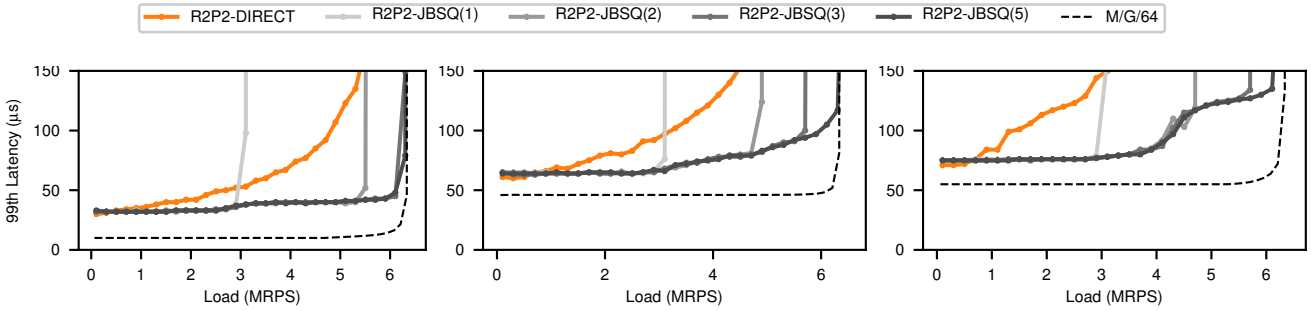
Also, for small sizes of this benchmark, `JBSQ(n=3)` is required, but also sufficient, to saturate the router.

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 2000B. 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 higher overheads of `nginx` operating as a `tcp` proxy (1) limit short-message performance by an order of magnitude and (2) the I/O bottleneck of HTTP reverse proxies limits goodput to the load balancers's 10Gbps NIC.

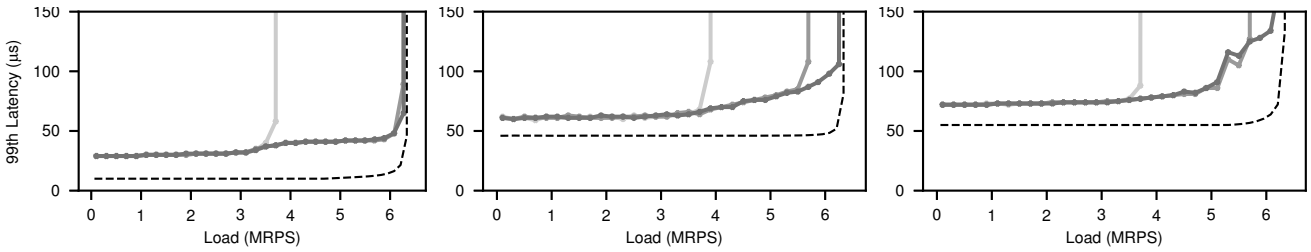
Latency vs. Throughput: Figure 6a uses a zero-cost ("echo") RPC service with 8-byte requests and responses, to measure the 99^{th} percentile tail latency as a function of the load for three configuration: the software middlebox, the Tofino router, the `nginx` baseline (Figure 6b only) and the direct configuration which bypasses any router.

Figure 6a shows the latency added by the router to be 5µs for the software middlebox and $< 1\mu s$ 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



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

Figure 7: Microbenchmarks using software middlebox. Service time $\bar{S} = 10\mu s$.



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

Figure 8: Microbenchmarks using Tofino router. Service time $\bar{S} = 10\mu s$.

packet headers in the dataplane. For Tofino, we observe a small increase in latency as a function load, which we attribute to the increase number of recirculation rounds.

Figure 6a also shows that the latency remains constant up to the point of saturation, which corresponds to 7 MRPS for the software middlebox (as discussed above). We were unable to characterize the maximal per-port capability of the Tofino ASIC running the R2P2 logic beyond > 8.5 MRPPS with tiny requests and replies, simply for lack of available machines.

Comparison of scheduling policies: Figure 6b uses a synthetic 25µs exponentially-distributed service time as workload. This is a coarse-grain workload and R2P2 with `JBSQ(3)` is capable of approximating the theoretically-optimal $M/M/64$ in terms of tail latency at low loads and in terms of throughput. `nginx`, even though it is configured to use 16 threads, is CPU-bound, and off by an order of magnitude in terms of both throughput (200KQPS) and tail latency ($\geq 300\mu s$). All R2P2 routed configurations outperform the direct connect model. `R2P2-RR` performs better than `R2P2-DIRECT` given the service time variability. `JBSQ(1)` limits throughput as it exposes the roundtrip latencies between worker and router but behaves similarly to a single-queue model until saturation. `JBSQ(n>1)` is required to further increase throughput, but can no longer guarantee tail-optimal decisions. Yet, the tail latency remains below $2\times$ the unloaded latency until nearly saturation. Note that `R2P2-JSQ` (not shown) performs similarly to `JBSQ(3)`.

Large responses: Figure 6c shows how `JBSQ` can be used to avoid IO bottlenecks, not just busy CPUs. We run a synthetic response-size RPC service with responses from bimodal distribution where 95% of the responses are 8 bytes, while 5% of them are 64 KB. This is an I/O bound workload. We observe that the proper `JBSQ(n)` policy can identify the workers with congested `Tx` queues, as control messages are sent after responses, and schedule accordingly to lower tail latency.

5.2 Synthetic Microbenchmarks

Figure 7 and Figure 8 evaluate `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 [49]. Figure 7 uses the software middlebox while Figure 8 uses the Tofino router. Requests and the responses are 8 bytes.

We first observe, for all 6 experiments, that all `JBSQ(n)` variants approximate the optimal single-queue approach until the saturation point for `JBSQ(1)`. We then observe (1) an increase in the tail latency as the system configuration trades off higher throughput against the use of a theoretically-

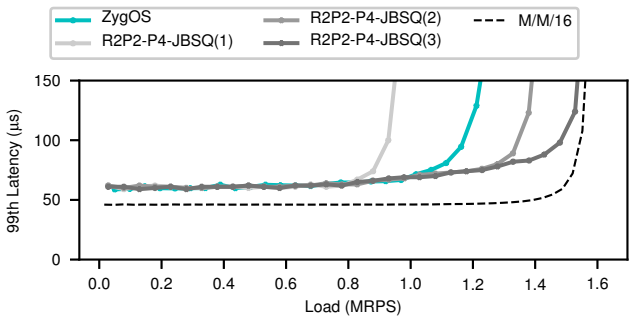


Figure 9: Comparison of R2P2 with the Zygos [72] work-conserving scheduler: Exponential workload with $\bar{S} = 10\mu s$.

optimal approach; (2) the `JBSQ(n)` policy must be configured with different values of n to saturate the throughput. The higher latency of the software middlebox typically requires $n+1$ outstanding requests over the Tofino router; this is consistent with the $\sim$5µs difference in latency; (3) workloads with greater dispersion (*e.g.,* the bimodal workload) require a larger value of n for the same $\bar{S}$. This is explained simply by the fact that the great majority of such requests are much smaller than $\bar{S}$. Nevertheless, `JBSQ(`$n \leq 5$`)` is sufficient to sufficient to saturate throughput in all 6 experiments.

5.3 Using R2P2 for server work conservation

We now demonstrate how the use of network-based load-balancing, *e.g.,* using R2P2, can be used to increase the efficiency in which a *single* server schedules tasks. For this, we compare the performance of Zygos [72], 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: Zygos does it within the server in a protocol-agnostic manner, whereas R2P2 implements the policy in the network.

Figure 9 compares ZygOS with `R2P2-JBSQ(n)` for the 10µs exponentially-distributed service time workload using a single Xeon server. For R2P2, each server core is configured with its own UDP port and we use the Tofino/P4 implementation. In this experiment, the lower bound is therefore determined by $M/M/16$. We observe that the throughput performance of Zygos is between `JBSQ(1)` and `JBSQ(2)`, and that `JBSQ(3)` achieves the maximum throughput with a tail latency that is only marginally higher than the lower bound. For a service-level objective set at 150µs, R2P2 with `JBSQ(3)` outperforms Zygos by 1.26×. This is possible as R2P2 server runs of 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.

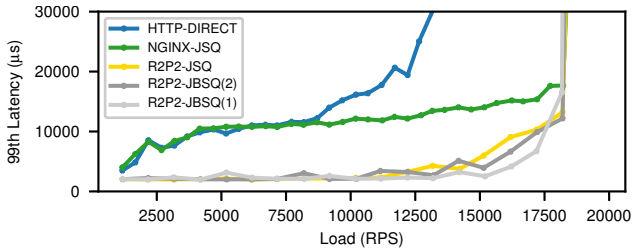


Figure 10: Lucene++ running on 16 16-threaded workers

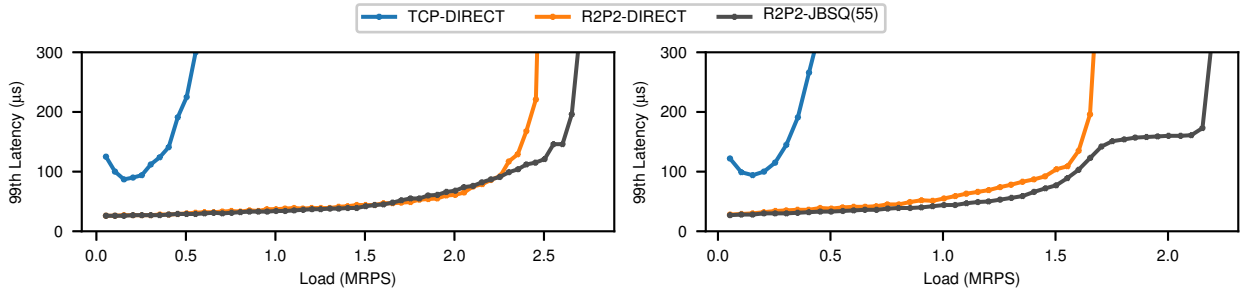
5.4 Lucene++

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++ [50], 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 [85], 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 [51]. 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-worker 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` which is a multi-queue model, has higher tail-latency. Then, we see that `NGINX-JSQ` and `R2P2-JBSQ(n)` deliver the same throughput; system and network protocol overheads are irrelevant for such coarse-grain workload. However, `R2P2-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 6× at 50% system load over `nginx`. Finally, we observe that `R2P2-JSQ`, despite having the similar system overheads as `JBSQ(n)`, still performs worse under high load given the increased service time variability of the workload.

5.5 Redis

Redis [74] 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



(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.

`sticky` R2P2 policy (see §3) to direct writes to the master node and we load-balance reads across the master and slave nodes. 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. Redis has sub-µs service times, and we configured adaptive batching on the servers. Thus, to achieve maximum throughput we had to increase the number of tokens to 55 per worker (`JBSQ(55)`).

Figure 11a shows that R2P2, for an SLO of 200µs at the 99^{th} percentile, achieves 5.85× better throughput for the USR workload over vanilla Redis because of reduced protocol and system overheads. Figure 11b increases the write percentage of the workload from 0.2% to 2%, which increases service time variability: `R2P2-DIRECT` has 4.66× better throughput than `TCP-DIRECT`. On top of this though, `R2P2-JBSQ` further improves throughput by 30%, for a total speedup of 6.1×.

6 Related Work

RPCs can be transported by different IP-based protocols including HTTP2 [10], QUIC [45], SCTP [78], DCCP [44], or similar research approaches [4, 31, 33, 35, 58] that identify the TCP limitations and optimize for flow-completion time. Libraries such as gRPC [34] and Thrift [80] abstract away the underlying transport stream into request-reply pairs. Load balancers proxy RPC protocols such as http in software [24, 61, 64] or in hardware [1, 17, 26, 54]. Unlike eRPC [41], R2P2 was defined directly with flexible routing in mind and similarly exposes an RPC-oriented API on both ends.

Load dispatching, direct or through load balancers, typically *pushes* requests to workers, requiring tail-mitigation techniques [18, 36] on top. In Join-Idle-Queue [49], 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 [20, 21, 42, 66, 71, 73, 86], typically operating with millsecond-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 [75] in combination with operating systems [25, 69] or dataplane [9, 39, 70] support. Zygos [72] is an intra-server, work-conserving scheduler for short tasks that relies 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 [48, 53, 65, 74]. MICA provide concurrent-read/exclusive-access (CREW) within a server [48] by offloading the routing decisions to the client, while hardware and software middleboxes [40, 47, 62] try to enhance the performance and functionality of key-value stores. RackOut extended the notion of CREW to rack-scale systems [63]. R2P2 provides a mechanisms for steering policies which can be used to implement CREW both within a single server and across the datacenter.

7 Conclusion

We revisit the requirements to support µs-scale RPCs across tiers of web-scale applications and propose to solve the problem by centralizing the scheduling decisions into a scalable router with a hybrid push/pull policy, rather than at the client or server ends. To achieve this, we design, implement and evaluate a transport protocol developed specifically for µs-scale RPCs that separates target selection from request and reply streaming. We show in particular that such a protocol, together with the join-bounded-shortest queue policy, can be implemented efficiently on a state-of-the-art programmable ASIC dataplane. Our approach outperforms standard load-balancing proxies by an order of magnitude in throughput and latency, achieves close to the theoretical optimal behavior of 10µs tasks, reduces the tail latency of websearch by 6× at 50% load, and increases the scalability of Redis in a master-slave configuration by more than 6.1×.

References

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

[2] ALIZADEH, M., GREENBERG, A. G., MALTZ, D. A., PADHYE, J., PATEL, P., PRABHAKAR, B., SENGUPTA, S., AND SRIDHARAN, M. Data center TCP (DCTCP). In *Proceedings of the ACM SIGCOMM 2010 Conference* (2010), pp. 63–74.

[3] ALIZADEH, M., KABBANI, A., EDSALL, T., PRABHAKAR, B., VAHDAT, A., AND YASUDA, M. 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)* (2012), pp. 253–266.

[4] ALIZADEH, M., YANG, S., SHARIF, M., KATTI, S., MCKEOWN, N., PRABHAKAR, B., AND SHENKER, S. pFabric: minimal near-optimal datacenter transport. In *Proceedings of the ACM SIGCOMM 2013 Conference* (2013), pp. 435–446.

[5] ATIKOGLU, B., XU, Y., FRACHTENBERG, E., JIANG, S., AND PALECZNY, M. 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* (2012), pp. 53–64.

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

[7] BARROSO, L. A., CLIDARAS, J., AND HÖLZLE, U. *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] BARROSO, L. A., DEAN, J., AND HÖLZLE, U. Web Search for a Planet: The Google Cluster Architecture. *IEEE Micro 23*, 2 (2003), 22–28.

[9] BELAY, A., PREKAS, G., PRIMORAC, M., KLIMOVIC, A., GROSSMAN, S., KOZYRAKIS, C., AND BUGNION, E. 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.

[10] BELSHE, M., PEON, R., AND THOMSON, M. Hypertext Transfer Protocol Version 2 (HTTP/2). RFC 7540 (Proposed Standard), May 2015.

[11] BIRRELL, A., AND NELSON, B. J. Implementing Remote Procedure Calls. *ACM Trans. Comput. Syst. 2*, 1 (1984), 39–59.

[12] P4 behavioral model - v2. https://github.com/p4lang/behavioral-model. Accessed on 30.04.2018.

[13] BOSSHART, P., DALY, D., GIBB, G., IZZARD, M., MCKEOWN, N., REXFORD, J., SCHLESINGER, C., TALAYCO, D., VAHDAT, A., VARGHESE, G., AND WALKER, D. P4: programming protocol-independent packet processors. *Computer Communication Review 44*, 3 (2014), 87–95.

[14] BRONSON, N., AMSDEN, Z., CABRERA, G., CHAKKA, P., DIMOV, P., DING, H., FERRIS, J., GIARDULLO, A., KULKARNI, S., LI, H. C., MARCHUKOV, M., PETROV, D., PUZAR, L., SONG, Y. J., AND VENKATARAMANI, V. TAO: Facebook's Distributed Data Store for the Social Graph. In *Proceedings of the 2013 USENIX Annual Technical Conference (ATC)* (2013), pp. 49–60.

[15] BUDIU, M., AND DODD, C. The P416 Programming Language. *Operating Systems Review 51*, 1 (2017), 5–14.

[16] CHO, I., JANG, K., AND HAN, D. Credit-Scheduled Delay-Bounded Congestion Control for Datacenters. In *Proceedings of the ACM SIGCOMM 2017 Conference* (2017), pp. 239–252.

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

[18] DEAN, J., AND BARROSO, L. A. The tail at scale. *Commun. ACM 56*, 2 (2013), 74–80.

[19] DECANDIA, G., HASTORUN, D., JAMPANI, M., KAKULAPATI, G., LAKSHMAN, A., PILCHIN, A., SIVASUBRAMANIAN, S., VOSSHALL, P., AND VOGELS, W. Dynamo: amazon's highly available key-value store. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP)* (2007), pp. 205–220.

[20] DELGADO, P., DIDONA, D., DINU, F., AND ZWAENEPOEL, W. Job-aware Scheduling in Eagle: Divide and Stick to Your Probes. In *Proceedings of the 2016 ACM Symposium on Cloud Computing (SOCC)* (2016), pp. 497–509.

[21] DELGADO, P., DINU, F., KERMARREC, A.-M., AND ZWAENEPOEL, W. Hawk: Hybrid Datacenter Scheduling. In *Proceedings of the 2015 USENIX Annual Technical Conference (ATC)* (2015), pp. 499–510.

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

[23] DRAGOJEVIC, A., NARAYANAN, D., CASTRO, M., AND HODSON, O. FaRM: Fast Remote Memory. In *Proceedings of the 11th Symposium on Networked Systems Design and Implementation (NSDI)* (2014), pp. 401–414.

[24] EISENBUD, D. E., YI, C., CONTAVALLI, C., SMITH, C., KONONOV, R., MANN-HIELSCHER, E., CILINGIROGLU, A., CHEYNEY, B., SHANG, W., AND HOSEIN, J. D. Maglev: A Fast and Reliable Software Network Load Balancer. In *Proceedings of the 13th Symposium on Networked Systems Design and Implementation (NSDI)* (2016), pp. 523–535.

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

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

[27] FÄRBER, F., CHA, S. K., PRIMSCH, J., BORNHÖVD, C., SIGG, S., AND LEHNER, W. SAP HANA database: data management for modern business applications. *SIGMOD Record 40*, 4 (2011), 45–51.

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

[29] 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.

[30] FLOYD, S., AND JACOBSON, V. Random early detection gateways for congestion avoidance. *IEEE/ACM Trans. Netw. 1*, 4 (1993), 397–413.

[31] FORD, B. Structured streams: a new transport abstraction. In *Proceedings of the ACM SIGCOMM 2007 Conference* (2007), pp. 361–372.

[32] FOX, A., AND BREWER, E. A. Harvest, Yield and Scalable Tolerant Systems. In *Proceedings of The 7th Workshop on Hot Topics in Operating Systems (HotOS-VII)* (1999), pp. 174–178.

[33] GAO, P. X., NARAYAN, A., KUMAR, G., AGARWAL, R., RATNASAMY, S., AND SHENKER, S. pHost: distributed near-optimal datacenter transport over commodity network fabric. In *Proceedings of the 2015 ACM Conference on Emerging Networking Experiments and Technology (CoNEXT)* (2015), pp. 1:1–1:12.

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

[35] HANDLEY, M., RAICIU, C., AGACHE, A., VOINESCU, A., MOORE, A. W., ANTICHI, G., AND WJCIK, M. Re-architecting datacenter networks and stacks for low latency and high performance. In *Proceedings of the ACM SIGCOMM 2017 Conference* (2017), pp. 29–42.

[36] HAO, M., LI, H., TONG, M. H., PAKHA, C., SUMINTO, R. O., STUARDO, C. A., CHIEN, A. A., AND GUNAWI, H. S. MittOS: Supporting Millisecond Tail Tolerance with Fast Rejecting SLO-Aware OS Interface. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)* (2017), pp. 168–183.

[37] Haproxy dsr. https://www.haproxy.com/blog/layer-4-load-balancing-direct-server-return-mode/.

[38] HAQUE, M. E., EOM, Y. H., HE, Y., ELNIKETY, S., BIANCHINI, R., AND MCKINLEY, K. S. 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)* (2015), pp. 161–175.

[39] JAMSHED, M. A., MOON, Y., KIM, D., HAN, D., AND PARK, K. mOS: A Reusable Networking Stack for Flow Monitoring Middleboxes. In *Proceedings of the 14th Symposium on Networked Systems Design and Implementation (NSDI)* (2017), pp. 113–129.

[40] JIN, X., LI, X., ZHANG, H., SOULÉ, R., LEE, J., FOSTER, N., KIM, C., AND STOICA, I. NetCache: Balancing Key-Value Stores with Fast In-Network Caching. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)* (2017), pp. 121–136.

[41] KALIA, A., KAMINSKY, M., AND ANDERSEN, D. G. Datacenter RPCs can be General and Fast. *CoRR abs/1806.00680* (2018).

[42] KARANASOS, K., RAO, S., CURINO, C., DOUGLAS, C., CHALIPARAMBIL, K., FUMAROLA, G. M., HEDDAYA, S., RAMAKRISHNAN, R., AND SAKALANAGA, S. Mercury: Hybrid Centralized and Distributed Scheduling in Large Shared Clusters. In *Proceedings of the 2015 USENIX Annual Technical Conference (ATC)* (2015), pp. 485–497.

[43] KOGIAS, M., AND BUGNION, E. Flow Control for Latency-Critical RPCs. In *Proceedings of the 2018 SIGCOMM Workshop on Kernel Bypassing Networks* (2018), KBNets'18, ACM, pp. 15–21.

[44] KOHLER, E., HANDLEY, M., AND FLOYD, S. Datagram Congestion Control Protocol (DCCP). RFC 4340 (Proposed Standard), Mar. 2006. Updated by RFCs 5595, 5596, 6335, 6773.

[45] LANGLEY, A., RIDDOCH, A., WILK, A., VICENTE, A., KRASIC, C., ZHANG, D., YANG, F., KOURANOV, F., SWETT, I., IYENGAR, J. R., BAILEY, J., DORFMAN, J., ROSKIND, J., KULIK, J., WESTIN, P., TENNETI, R., SHADE, R., HAMILTON, R., VASILIEV, V., CHANG, W.-T., AND SHI, Z. The QUIC Transport Protocol: Design and Internet-Scale Deployment. In *Proceedings of the ACM SIGCOMM 2017 Conference* (2017), pp. 183–196.

[46] LE BOUDEC, J.-Y. *Performance Evaluation of Computer and Communication Systems*. EPFL Press, Lausanne, Switzerland, 2010.

[47] LI, X., SETHI, R., KAMINSKY, M., ANDERSEN, D. G., AND FREEDMAN, M. J. Be Fast, Cheap and in Control with SwitchKV. In *Proceedings of the 13th Symposium on Networked Systems Design and Implementation (NSDI)* (2016), pp. 31–44.

[48] LIM, H., HAN, D., ANDERSEN, D. G., AND KAMINSKY, M. MICA: A Holistic Approach to Fast In-Memory Key-Value Storage. In *Proceedings of the 11th Symposium on Networked Systems Design and Implementation (NSDI)* (2014), pp. 429–444.

[49] LU, Y., XIE, Q., KLIOT, G., GELLER, A., LARUS, J. R., AND GREENBERG, A. G. Join-Idle-Queue: A novel load balancing algorithm for dynamically scalable web services. *Perform. Eval. 68*, 11 (2011), 1056–1071.

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

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

[52] MEISNER, D., SADLER, C. M., BARROSO, L. A., WEBER, W.-D., AND WENISCH, T. F. Power management of online data-intensive services. In *Proceedings of the 38th International Symposium on Computer Architecture (ISCA)* (2011), pp. 319–330.

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

[54] MIAO, R., ZENG, H., KIM, C., LEE, J., AND YU, M. SilkRoad: Making Stateful Layer-4 Load Balancing Fast and Cheap Using Switching ASICs. In *Proceedings of the ACM SIGCOMM 2017 Conference* (2017), pp. 15–28.

[55] MITTAL, R., SHPINER, A., PANDA, A., ZAHAVI, E., KRISHNAMURTHY, A., RATNASAMY, S., AND SHENKER, S. Revisiting network support for RDMA. In *Proceedings of the ACM SIGCOMM 2018 Conference* (2018), pp. 313–326.

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

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

[58] MONTAZERI, B., LI, Y., ALIZADEH, M., AND OUSTERHOUT, J. K. Homa: a receiver-driven low-latency transport protocol using network priorities. In *Proceedings of the ACM SIGCOMM 2018 Conference* (2018), pp. 221–235.

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

[60] Nginx dsr. https://www.nginx.com/blog/ip-transparency-direct-server-return-nginx-plus-transpare

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

[62] NISHTALA, R., FUGAL, H., GRIMM, S., KWIATKOWSKI, M., LEE, H., LI, H. C., MCELROY, R., PALECZNY, M., PEEK, D., SAAB, P., STAFFORD, D., TUNG, T., AND VENKATARAMANI, V. Scaling Memcache at Facebook. In *Proceedings of the 10th Symposium on Networked Systems Design and Implementation (NSDI)* (2013), pp. 385–398.

[63] NOVAKOVIC, S., DAGLIS, A., BUGNION, E., FALSAFI, B., AND GROT, B. The Case for RackOut: Scalable Data Serving Using Rack-Scale Systems. In *Proceedings of the 2016 ACM Symposium on Cloud Computing (SOCC)* (2016), pp. 182–195.

[64] OLTEANU, V. A., AGACHE, A., VOINESCU, A., AND RAICIU, C. Stateless Datacenter Load-balancing with Beamer. In *Proceedings of the 15th Symposium on Networked Systems Design and Implementation (NSDI)* (2018), pp. 125–139.

[65] OUSTERHOUT, J. K., GOPALAN, A., GUPTA, A., KEJRIWAL, A., LEE, C., MONTAZERI, B., ONGARO, D., PARK, S. J., QIN, H., ROSENBLUM, M., RUMBLE, S. M., STUTSMAN, R., AND YANG, S. The RAMCloud Storage System. *ACM Trans. Comput. Syst. 33*, 3 (2015), 7:1–7:55.

[66] OUSTERHOUT, K., WENDELL, P., ZAHARIA, M., AND STOICA, I. Sparrow: distributed, low latency scheduling. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)* (2013), pp. 69–84.

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

[68] P4-16 compiler reference implementation. https://github.com/p4lang/p4c. Accessed on 30.04.2018.

[69] PESTEREV, A., STRAUSS, J., ZELDOVICH, N., AND MORRIS, R. T. Improving network connection locality on multicore systems. In *Proceedings of the 2012 EuroSys Conference* (2012), pp. 337–350.

[70] PETER, S., LI, J., ZHANG, I., PORTS, D. R. K., WOOS, D., KRISHNAMURTHY, A., ANDERSON, T. E., AND ROSCOE, T. Arrakis: The Operating System Is the Control Plane. *ACM Trans. Comput. Syst. 33*, 4 (2016), 11:1–11:30.

[71] POWER, R., AND LI, J. Piccolo: Building Fast, Distributed Programs with Partitioned Tables. In *Proceedings of the 9th Symposium on Operating System Design and Implementation (OSDI)* (2010), pp. 293–306.

[72] PREKAS, G., KOGIAS, M., AND BUGNION, E. ZygOS: Achieving Low Tail Latency for Microsecond-scale Networked Tasks. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)* (2017), pp. 325–341.

[73] RASLEY, J., KARANASOS, K., KANDULA, S., FONSECA, R., VOJNOVIC, M., AND RAO, S. Efficient queue management for cluster scheduling. In *Proceedings of the 2016 EuroSys Conference* (2016), pp. 36:1–36:15.

[74] Redis. `https://redis.io/`.

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

[76] SALTZER, J. H., REED, D. P., AND CLARK, D. D. End-To-End Arguments in System Design. *ACM Trans. Comput. Syst. 2*, 4 (1984), 277–288.

[77] SCHROEDER, B., WIERMAN, A., AND HARCHOL-BALTER, M. Open Versus Closed: A Cautionary Tale. In *Proceedings of the 3rd Symposium on Networked Systems Design and Implementation (NSDI)* (2006).

[78] STEWART, R. Stream Control Transmission Protocol. RFC 4960 (Proposed Standard), Sept. 2007. Updated by RFCs 6096, 6335, 7053.

[79] STONEBRAKER, M., MADDEN, S., ABADI, D. J., HARIZOPOULOS, S., HACHEM, N., AND HELLAND, P. 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)* (2007), pp. 1150–1160.

[80] Apache thrift. `https://thrift.apache.org/`.

[81] TU, S., ZHENG, W., KOHLER, E., LISKOV, B., AND MADDEN, S. Speedy transactions in multicore in-memory databases. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)* (2013), pp. 18–32.

[82] Voltdb. `https://www.voltdb.com/`.

[83] WEI, X., SHI, J., CHEN, Y., CHEN, R., AND CHEN, H. Fast in-memory transaction processing using RDMA and HTM. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)* (2015), pp. 87–104.

[84] WIERMAN, A., AND ZWART, B. Is Tail-Optimal Scheduling Possible? *Operations Research 60*, 5 (2012), 1249–1257.

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

[86] ZAHARIA, M., XIN, R. S., WENDELL, P., DAS, T., ARMBRUST, M., DAVE, A., MENG, X., ROSEN, J., VENKATARAMAN, S., FRANKLIN, M. J., GHODSI, A., GONZALEZ, J., SHENKER, S., AND STOICA, I. Apache Spark: a unified engine for big data processing. *Commun. ACM 59*, 11 (2016), 56–65.

[87] ZHANG, H., DONG, M., AND CHEN, H. 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)* (2016), pp. 167–180.

[88] ZHU, Y., ERAN, H., FIRESTONE, D., GUO, C., LIPSHTEYN, M., LIRON, Y., PADHYE, J., RAINDEL, S., YAHIA, M. H., AND ZHANG, M. Congestion Control for Large-Scale RDMA Deployments. In *Proceedings of the ACM SIGCOMM 2015 Conference* (2015), pp. 523–536.