Remote Procedure Call as a Managed System Service

Jingrong Chen, Yongji Wu, and Shihan Lin, *Duke University;* Yechen Xu, *Shanghai Jiao Tong University;* Xinhao Kong, *Duke University;* Thomas Anderson, *University of Washington;* Matthew Lentz, Xiaowei Yang, and Danyang Zhuo, *Duke University*

https://www.usenix.org/conference/nsdi23/presentation/chen-jingrong

This paper is included in the
Proceedings of the 20th USENIX Symposium on
Networked Systems Design and Implementation.

April 17–19, 2023 • Boston, MA, USA

978-1-939133-33-5

Open access to the Proceedings of the
20th USENIX Symposium on Networked
Systems Design and Implementation
is sponsored by

جامعة الملك عبدالله
للعلوم والتقنية
King Abdullah University of
Science and Technology

Remote Procedure Call as a Managed System Service

Jingrong Chen[1,*] Yongji Wu[1,*] Shihan Lin[1] Yechen Xu[3] Xinhao Kong[1]
Thomas Anderson[2] Matthew Lentz[1] Xiaowei Yang[1] Danyang Zhuo[1]

[1]*Duke University* [2]*University of Washington* [3]*Shanghai Jiao Tong University*

Abstract

Remote Procedure Call (RPC) is a widely used abstraction for cloud computing. The programmer specifies type information for each remote procedure, and a compiler generates stub code linked into each application to marshal and unmarshal arguments into message buffers. Increasingly, however, application and service operations teams need a high degree of visibility and control over the flow of RPCs between services, leading many installations to use sidecars or service mesh proxies for manageability and policy flexibility. These sidecars typically involve inspection and modification of RPC data that the stub compiler had just carefully assembled, adding needless overhead. Further, upgrading diverse application RPC stubs to use advanced hardware capabilities such as RDMA or DPDK is a long and involved process, and often incompatible with sidecar policy control.

In this paper, we propose, implement, and evaluate a novel approach, where RPC marshalling and policy enforcement are done as a system service rather than as a library linked into each application. Applications specify type information to the RPC system as before, while the RPC service executes policy engines and arbitrates resource use, and then marshals data customized to the underlying network hardware capabilities. Our system, mRPC, also supports live upgrades so that both policy and marshalling code can be updated transparently to application code. Compared with using a sidecar, mRPC speeds up a standard microservice benchmark, DeathStarBench, by up to 2.5× while having a higher level of policy flexibility and availability.

1 Introduction

Remote Procedure Call (RPC) is a fundamental building block of distributed systems in modern datacenters. RPC allows developers to build networked applications using a simple and familiar programming model [10], supported by several popular libraries such as gRPC [26], Thrift [84], and eRPC [39]. The RPC model has been widely adopted in distributed data stores [19, 41, 83], network file systems [24, 80], consensus protocols [68], data-analytic frameworks [2, 12, 16, 25, 55, 82, 94, 98], cluster schedulers and orchestrators [30, 50], and machine learning systems [1, 65, 72]. Google found that roughly 10% of its datacenter CPU cycles are spent just executing gRPC library code [42]. Because of its importance, improving RPC performance has long been a major topic of research [7, 8, 10, 14, 39, 52, 63, 81, 87, 95, 96].

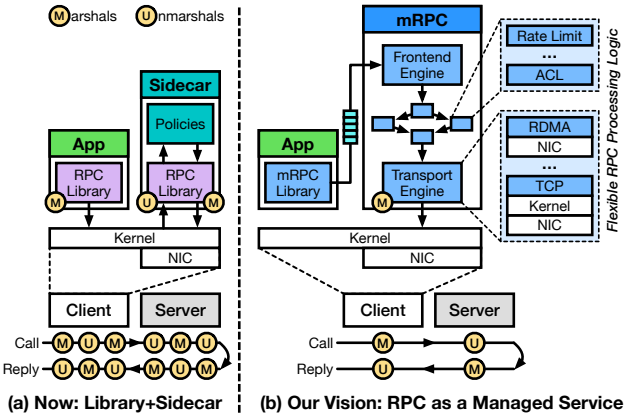


Figure 1: Architectural comparison between current (RPC library + sidecar) and our proposed (RPC as a managed service) approaches.

Recently, application and network operations teams have found a need for rapid and flexible visibility and control over the flow of RPCs in datacenters. This includes monitoring and control of the performance of specific types of RPCs [62], prioritization and rate limiting to meet application-specific performance and availability goals, dynamic insertion of advanced diagnostics to track user requests across a network of microservices [22], and application-specific load balancing to improve cache effectiveness [6], to name a few.

The typical architecture is to enforce policies in a sidecar—a separate process that mediates the network traffic of the application RPC library (Figure 1a). This is often referred to as a service mesh. A number of commercial products have been developed to meet the need for sidecar RPC proxies, such as

*Jingrong Chen and Yongji Wu contributed equally.

Envoy [18], Istio [32], HAProxy [29], Linkerd [53], Nginx [67], and Consul [15]. Although some policies could theoretically be supported by a feature-rich RPC runtime linked in with each application, that can slow deployment—Facebook recently reported that it can take *months* to fully roll out changes to one of its application communication libraries [21]. One use case that requires rapid deployment is to respond to a new application security threat, or to diagnose and fix a critical user-visible failure. Finally, many policies are mandatory rather than discretionary—the network operations team may not be able to trust the library code linked into an application. Example mandatory security policies include access control, authentication/encryption [15], and prevention of known exploits in widely used network protocols such as RDMA [79].

Although using a sidecar for policy management is functional and secure, it is also inefficient. The application RPC library marshals RPC parameters at runtime into a buffer according to the type information provided by the programmer. This buffer is sent through the operating system network stack and then forwarded back up to the sidecar, which typically needs to parse and unwrap the network, virtualization, and RPC headers, often looking inside the packet payload to correctly enforce the desired policy. It then re-marshals the data for transport. Direct application-level access to network hardware such as RDMA or DPDK offers high performance but precludes sidecar policy control. Similarly, network interface cards are increasingly sophisticated, but it is hard for applications or sidecars to take advantage of those new features, because marshalling is done too high up in the network stack. Any change to the marshalling code requires recompiling and rebooting each application and/or the sidecar, hurting end-to-end availability. In short, existing solutions can provide good performance, or flexible and enforceable policy control, but not both.

In this paper, we propose a new approach, called RPC as a managed service, to address these limitations. Instead of separating marshalling and policy enforcement across different domains, we combine them into a single privilege and trusted system service (Figure 1b) so that marshalling is done **after** policy processing. In our prototype, mRPC for managed RPC, the privileged RPC service runs at user level communicating with the application through shared memory regions [4,8,58]. However, mRPC could also be integrated directly into the operating system kernel with a dynamically replaceable kernel module [61].

Our goals are to be fast, support flexible policies, and provide high availability for applications. To achieve this, we need to address several challenges. First, we need to decouple marshalling from the application RPC library. Second, we need to design a new policy enforcement mechanism to process RPCs efficiently and securely, without incurring additional marshalling overheads. Third, we need to provide a way for operators to specify/change policies and even change the underlying transport implementation without disrupting running applications.

We implement mRPC, the first RPC framework that follows the RPC as a managed service approach. Our results show that mRPC speeds up DeathStarBench [23] by up to 2.5×, in terms of mean latency, compared with combining state-of-art RPC libraries and sidecars, i.e., gRPC and Envoy, using the same transport mechanism. Larger performance gains are possible by fully exploiting network hardware capabilities from within the service. In addition, mRPC allows for live upgrades of its components while incurring negligible downtime for applications. Applications do not need to be re-compiled or rebooted to change policies or marshalling code. mRPC has three important limitations. First, data structures passed as RPC arguments must be allocated on a special shared-memory heap. Second, while we use a language-independent protocol for specifying RPC type signatures, our prototype implementation currently only works with applications written in Rust. Finally, our stub generator is not as fully featured as gRPC.

In this paper, we make the following contributions:

- A novel RPC architecture that decouples marshalling/unmarshalling from RPC libraries to a centralized system service.
- An RPC mechanism that applies network policies and observability features with both security and low performance overhead, i.e., with minimal data movement and no redundant (un)marshalling. The mechanism supports live upgrade of RPC bindings, policies, transport, and marshalling without disrupting running applications.
- A prototype implementation of mRPC along with an evaluation on both synthetic workloads and real applications.

2 Background

In this section, we discuss the current RPC library architecture. We then discuss the emerging need for manageability and how manageability is implemented with existing RPC libraries.

2.1 Remote Procedure Call

To use RPC, a developer defines the relevant service interfaces and message types in a schema file (e.g., gRPC `.proto` file). A protocol compiler will translate the schema into program stubs that are directly linked with the client and server applications. To issue an RPC at runtime, the application simply calls the corresponding function provided by the stub; the stub is responsible for marshalling the request arguments and interacting with the transport layer (e.g., TCP/IP sockets or RDMA verbs). The transport layer delivers the packets to the remote server, where the stub unmarshals the arguments and dispatches the RPC request to a thread (eventually replying back to the client). We refer to this approach as *RPC-as-a-library*, since all RPC functionality is included in user-space libraries that are linked with each application. Even though the first RPC implementation [10] dates back to the 1980s, modern RPC frameworks (e.g., gRPC [26], eRPC [39], Thrift [84]) still follow this same approach.

A key design goal for RPC frameworks is efficiency. Google and Facebook have built their own efficient RPC frameworks, gRPC and Apache Thrift. Although primarily focused on portability and interoperability, gRPC includes many efficiency-related features, such as supporting binary payloads. Academic researchers have studied various ways to improve RPC efficiency, including optimizing the network stack [45,69,99], software hardware co-design [39, 41], and overload control [14].

As network link speeds continue to scale up [77], RPC overheads are likely to become even more salient in the future. This has led some researchers to advocate for direct application access to network hardware [5, 39, 73, 99], e.g., with RDMA or DPDK. Although low overhead, kernel bypass is largely incompatible with the need for flexible and enforceable layer 7 policy control, as we discuss next. In practice, multiple security weaknesses in RDMA hardware have led most cloud vendors to opt against providing direct access to RDMA by untrusted applications [48, 49, 58, 79, 95, 101].

2.2 The Need for Manageability

As RPC-based distributed applications scale to large, complex deployment scenarios, there is an increasing need for improved manageability of RPC traffic. We classify management needs into three categories: **1) Observability:** Provide detailed telemetry, which enables developers to diagnose and optimize application performance. **2) Policy Enforcement:** Allow operators to apply custom policies to RPC applications and services (e.g., access control, rate limits, encryption). **3) Upgradability:** Support software upgrades (e.g., bug fixes and new features) while minimizing downtime to applications.

One natural question to ask is: *is it possible to add these properties without changing existing RPC libraries?* For observability and policy enforcement, the state-of-the-art solution is to use a sidecar (e.g., Envoy [18] or Linkerd [53]). A sidecar is a standalone process that intercepts every packet an application sends, reconstructing the application-level data (i.e., RPC), and applying policies or enabling observability. However, using a sidecar introduces substantial performance overhead, due to redundant RPC (un)marshalling. This RPC (un)marshalling, for example, in gRPC+Envoy, including HTTP framing and protobuf encoding, accounts for 62-73% overhead in the end-to-end latency [102]. In our evaluation (§7), using a sidecar increases the 99th percentile RPC latency by 180% and decreases the bandwidth by 44%. Figure 1a shows the (un)marshalling steps invoked as an RPC traverses from a client to a server and back. Using a sidecar triples the number of (un)marshalling steps (from 4 to 12). In addition, the sidecar approach is largely incompatible with the emerging trend of efficient application-level access to network hardware. Using sidecars means data buffers have to be copied between the application and sidecars, reducing the benefits of having zero-copy kernel-bypass access to the network.

Finally, using sidecars with application RPC libraries does not completely solve the upgradability issue. While policy can often be changed dynamically (depending on the feature set of the sidecar implementation), marshalling and transport code is harder to change. To fix a bug in the underlying RPC library, or merely to upgrade the code to take advantage of new hardware features, we need to recompile the entire application (and sidecar) with the patched RPC library and reboot. gRPC has a monthly or two-month release cycle for bug fixes and new features [27]. Any scheduled downtime has to be communicated explicitly to the users of the application or has to be masked using replication; either approach can lead to complex application life-cycle management issues.

We do not see much hope in continuing to optimize this RPC library and sidecar approach for two reasons. First, a strong coupling exists between a traditional RPC library and each application. This makes upgrading the RPC library without stopping the application difficult, if not impossible. Second, there is only weak or no coupling between an RPC library and a sidecar. This prevents the RPC library and the sidecar from cross-layer optimization.

Instead, we argue for an alternative architecture in which RPC is provided *as a managed service*. By decoupling RPC logic, e.g., (un)marshalling, transport interface, from the application, the service can simultaneously provide high performance, policy flexibility, and zero-downtime upgrades.

3 Overview

Our system, mRPC, realizes the *RPC-as-a-managed-service* abstraction while maintaining similar end-to-end semantics as traditional RPC libraries (e.g., gRPC, Thrift). The goals for mRPC are to be fast, support flexible policy enforcement, and provide high availability for applications.

Figure 2 shows a high-level overview of the mRPC architecture and workflow, breaking it down into three major phases: initialization, runtime, and management. The mRPC service runs as a non-root, user-space process with access to the necessary network devices and a shared-memory region for each application. In each of the phases, we focus on the view of a single machine that is running both the RPC client application and the mRPC service. The RPC server may also run alongside an mRPC service. In this case, mRPC-specific marshalling can be used. However, we also support flexible marshalling to enable mRPC applications to interact with external peers using well-known formats (e.g., gRPC). In our evaluation, we focus on cases where both the client and server employ mRPC.

The initialization phase extends from building the application to how the application binds to a specific RPC interface. (1) Similar to gRPC, users define a protocol schema. The mRPC schema compiler uses this to generate stub code to include in their application. We illustrate this using a key-value storage service with a single `Get` function. (2) When the application is deployed, it connects with the mRPC service running on the same machine and specifies the protocol(s) of interest, which are maintained by the generated stub. (3) The mRPC service also uses the protocol schema to generate,

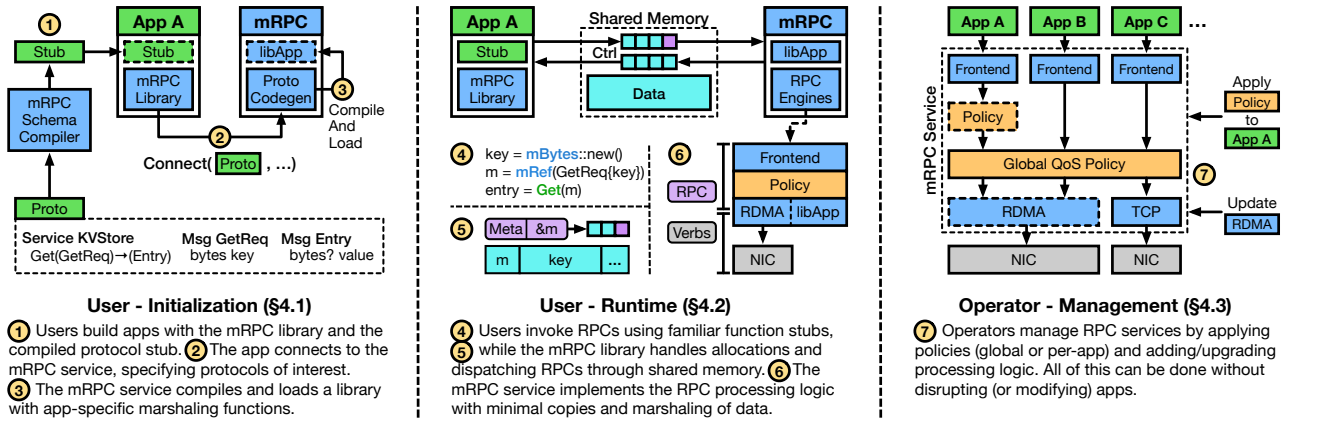


Figure 2: Overview of the mRPC workflow from the perspective of the users (and their applications) as well as infrastructure operators.

compile, and dynamically load a protocol-specific library containing the marshalling and unmarshalling code for that application's schemas[2]. This *dynamic binding* is a key enabler for mRPC to act as a long-running service, handling arbitrary applications (and their RPC schemas). [3]

At this point, we enter the runtime phase in which the application begins to invoke RPCs. Our approach uses *shared memory* between the application and mRPC, containing both control queues as well as a data buffer. (4) The application protocol stub produced by the mRPC protocol compiler can be called like a traditional RPC interface, with the exception that data structures passed as arguments or as return values must be allocated on a special heap in the shared data buffer. As an example, we show an excerpt of Rust-like pseudocode for invoking the `Get` function. (5) Internally, the stub and mRPC library manage RPC calls and replies in the control queues along with allocations and deallocations in the data buffer. (6) The mRPC service operates over the RPCs through modular *engines* that are composed to implement the per-application *datapaths* (i.e., sequence of RPC processing logic); each engine is responsible for one type of task (e.g., application interface, rate limiting, transport interface). Engines do not contain execution contexts, but are rather scheduled by *runtimes* in mRPC that correspond to kernel-level threads; during their execution, engines read from input queues, perform work, and enqueue outputs. External-facing engines (i.e., frontend, transport) use asynchronous control queues, while all other engines are executed synchronously by a runtime. Application control queues are contained in shared memory with the mRPC service.

This architecture, along with dynamic binding, enables mRPC to *operate over RPCs rather than packets*, avoiding the high overhead of traditional sidecar-based approaches. Additionally, the modular design of mRPC's processing logic enables mRPC to take advantage of fast network hardware (e.g., RDMA and smartNICs) in a manner that is transparent to the application. A key challenge, which we will address in §4.2, is how to securely enforce operator policies over RPCs in shared memory while minimizing data copies.

Finally, mRPC aims to improve the manageability of RPCs by infrastructure operators. Here, we zoom out to focus on the processing logic across all applications served by an mRPC service. (7) Operators may wish to apply a number of different policies to RPCs made by applications, whether on an individual basis (e.g., rate limiting, access control) or globally across applications (e.g., QoS). mRPC allows operators to add, remove, update, or reconfigure policies at runtime. This flexibility extends beyond policies to include those responsible for interacting with the network hardware. A key challenge, which we will address in §4.3, is in supporting the *live upgrade* of mRPC engines without interrupting running applications (and while managing engines sharing memory queues).

4 Design

In this section, we describe how mRPC provides dynamic binding, efficient policy and observability support, live upgrade, and security.

4.1 Dynamic RPC Binding

Applications have different RPC schemas, which ultimately decide how an RPC is marshalled. In the traditional RPC-as-a-library approach, a protocol compiler generates the marshalling code, which is linked into the application. In our design, the mRPC service is responsible for marshalling, which means that the application-specific marshalling code needs to be decoupled from an RPC library and run inside the mRPC service itself. Failing to ensure this separation would allow arbitrary code execution by a malicious user.

Applications directly submit the RPC schema (and not marshalling code) to the mRPC service. The mRPC service generates the corresponding marshalling code, then compiles and dynamically loads the library. Thus, we rely on our mRPC service code generator to produce the correct marshalling code

[2] Note that such libraries may be prefetched and/or cached to optimize the startup time.

[3] The dashed box of "Stub" and "libApp" means they are generated code.

for *any* user-provided RPC schema. For the initial handshake between an RPC client and an RPC server, the two mRPC services check that the provided RPC schemas match, and if not, the client's connection is rejected.

There are three remaining questions. First, **what are the responsibilities of the in-application user stub and mRPC library?** In mRPC, applications rely on user stubs to implement the abstraction as specified in their RPC schema. This means we still need to generate the glue code to maintain the traditional application programming interface. Our solution is to provide a separate protocol schema compiler, which is untrusted and run by application developers, to generate the user stub code that does not involve marshalling and transport. The application RPC stub (with the help of the mRPC library) creates a message buffer that contains the metadata of the RPC, with typed pointers to the RPC arguments, on the shared memory heap. The message is placed on a shared memory queue, which will be processed by the mRPC service. The receiving side works in a similar way.

Second, **does this approach increase RPC connect/bind time?** Implemented naively, this design will increase the RPC connect/bind time because the mRPC service has to compile the RPC schema and load the resulting marshalling library when an RPC client first connects to a corresponding server (or equivalently when an RPC server binds to the service). However, this latency is not fundamental to our design, and we can mitigate it in the following way. The mRPC service accepts RPC schemas before booting an application, as a form of prefetching. Given a schema, it compiles and caches the marshalling code. At the time of RPC connect/bind, the mRPC service simply performs a cache lookup based on the hash of the RPC schema. If it exists within the cache, the mRPC service will load the associated library; otherwise, the mRPC service will invoke the compiler to generate (and subsequently cache) the library. This reduces the connect/bind time from several seconds to several milliseconds.

Third, **when new applications arrive, do existing applications face downtime?** The multi-threaded mRPC service is a single process that serves many RPC applications; however, the marshalling engines for different RPC applications are not shared. They are in different memory addresses and can be (un)loaded independently. We will describe in §4.3 how to load/unload engines without disrupting running applications.

4.2 Efficient RPC Policy Enforcement and Observability

We have one key idea to allow efficient RPC policy enforcement and observability: senders should marshal once (as late as possible), while receivers should unmarshal once (as early as possible). On the sender side, we want to support policy enforcement and observability directly over RPCs from the application, and then marshal the RPC into packets. The receiver side is similar: packets should be unmarshalled into RPCs, applying policy and observability operations, and then delivered directly to the application. Compared to the traditional RPC-as-a-library approach with sidecars, this eliminates the redundant (un)marshalling steps (see Figure 1).

Data: DMA-capable shared memory heaps. Our design is centered around a dedicated shared memory heap between each application and the mRPC service. (Note that this heap is not shared across applications.) Applications directly create data structures, which may be used in RPC arguments, in a shared memory heap with the help of the mRPC library. Each application has a separate shared memory region, which provides isolation between (potentially mutually distrusting) applications. The mRPC library also includes a standard slab allocator for managing object allocation on this shared memory. If there is insufficient space within the shared memory, the slab allocator will request additional shared memory from the mRPC service and then map it into the application's address space. The mRPC service has access to the shared memory heap, allowing it to execute RPC processing logic over the application's RPCs, but also maintains a private memory heap for necessary copies.

Figure 3 shows an example workflow that includes access control for a key-value store service. Having the data structures directly in the shared memory allows an application to provide pointers to data, rather than the data itself, when submitting RPCs to the mRPC service. We call the message sent from an application to the mRPC service an *RPC descriptor*. If there are multiple RPC arguments, the RPC descriptor points to an array of pointers (each pointing to a different argument on the heap).

Let us say we have an ACL policy that rejects an RPC if the key matches a certain string. The mRPC service first copies the argument (i.e., `key`), as well as all parental data structures (i.e., `GetReq`), onto its private heap. This is to prevent time-of-use-to-time-of-check (TOCTOU) attacks. Since applications have access to DMA-capable shared memory at all times, an application could modify the content in the memory while the mRPC service is enforcing policies. Copying arguments is a standard mitigation technique, similar to how OS kernels prevents TOCTOU attacks by copying system call arguments from user- to kernel-space. This copying only needs to happen if the policy behavior is based on the content of the RPC. We demonstrate in §7.2 that even with such copying, mRPC's overhead for an ACL policy is much lower than gRPC + Envoy. The RPC descriptor is modified so that the pointer to the copied argument now points to the private heap. On the receiver side, the TOCTOU attack is not relevant, but we need to take care not to place RPCs directly in shared memory. If there is a receive-side policy that depends on RPC argument values, the mRPC service first receives the RPC data into a private heap; it copies the RPC data into the shared heap after policy processing. This prevents the application from reading RPC data that should have been dropped or modified by the policies. Note that we can bypass this copy when processing does not depend on RPC argument values (e.g., rate limits). During ACL policy enforcement, the RPC is dropped if the `key` argument is contained in a blocklist. Note that if an RPC is dropped, any further processing logic is never executed (including marshalling operations).

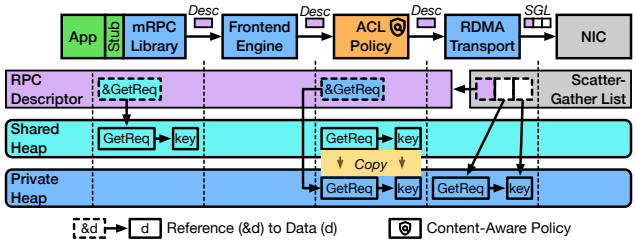


Figure 3: Overview of memory management in mRPC. Shows an example for the `Get` RPC that includes a content-aware ACL policy.

Finally, at the end of the processing logic, the transport adapter engine executes. mRPC currently supports two types of transport: TCP and RDMA. For TCP, mRPC uses the standard, kernel-provided scatter-gather (`iovec`) socket interface. For RDMA, mRPC uses the scatter-gather `verb` interface, allowing the NIC to directly interact with buffers on the shared (or private) memory heaps containing the RPC metadata and arguments. For both TCP and RDMA, mRPC provides disjoint memory blocks to the transport layer directly, eliminating excessive data movements.[4]

Control: Shared-memory queues. To facilitate efficient communication between an application and the mRPC service, we use shared memory control queues. mRPC allocates two unidirectional queues for sending and receiving requests from an application to the mRPC service. The requests contain RPC descriptors, which reference arguments on the shared memory heap. The mRPC service always copies the RPC descriptors applications put in the sending queue to prevent TOCTOU attacks. mRPC provides two options to poll the queues: 1) busy polling, and 2) `eventfd`-based adaptive polling. In busy polling, both the application-side mRPC library and the mRPC service busy poll on their ends of the queues. In the `eventfd` approach, the mRPC library and the mRPC service sends event notifications after enqueuing to an empty queue. After receiving a notification, the queue is drained (performing the necessary work) before subsequently waiting on future events. The `eventfd` approach saves CPU cycles when queues are empty. Other alternative solutions may involve dynamically scaling up (or down) the number of threads used to busy poll by the mRPC service; however, we chose the `eventfd` approach for its simplicity. In our evaluation, we use busy polling for RDMA and eventfd-based adaptive polling for TCP.

Memory management. We provide a memory allocator in the mRPC library for applications to directly allocate RPC data structures to be sent on a shared memory heap. The allocator invokes the mRPC service to allocate shared memory regions on behalf of the application (similar to how a standard heap manager calls `mmap` or `sbrk` to allocate memory from an OS kernel). We need to use a specialized memory allocator for RPC messages (and their arguments), since RPCs are shared between three entities: the application, the mRPC service, and the NIC. A memory block is safe to be reclaimed only when it will no longer be accessed by any entity.

We adopt a notification-based mechanism for memory management. On the sender side, the outgoing messages are managed by the mRPC library within the application. On the receiver side, the incoming messages are managed by the mRPC service. When the application no longer accesses a memory block occupied by outgoing messages, the memory block will not be reclaimed until the library receives a notification from mRPC service that the corresponding messages are already sent successfully through the NIC (similar to how zero-copy sockets work in Linux). Incoming messages are put in buffers on a separate read-only shared heap. The receiving buffers can be reclaimed when the application finishes processing (e.g., when the RPC returns). To support reclamation of receive buffers, the mRPC library notifies the mRPC service when specific messages are no longer in use by the application. Notifications for multiple RPC messages are batched to improve performance. If the receiver application code wishes to preserve or modify the incoming data, it must make an explicit copy. Although this differs from traditional RPC semantics, in our implementation of Masstree and DeathStarBench we found no examples where the extra copy was necessary.

Cross-datapath policy engines. mRPC supports engines that operate over multiple datapaths, which may span multiple applications. For instance, any global policy (e.g., QoS) will need to operate over all datapaths (see §5). For this type of engine, we instantiate replicas of the engine for each datapath that it applies to. Replicas can choose to either communicate through shared state, which requires managing contention across runtimes, or support runtime-local state that is contention-free.

4.3 Live Upgrades

Although our modular engine design for the mRPC service is similar to Snap [58] and Click [47], we arrive at very different designs for upgrades. Click does not support live upgrades, while Snap executes the upgraded process to run alongside the old process. The old process serializes the engine states, transfers them to the new process, and the new process restarts them. This means that even changing a single line of code within a single Snap engine requires a complete restart for all Snap engines. This design philosophy is fundamentally not compatible with mRPC, as we need to deal with new applications arriving with different RPC schemas, and thus our upgrades are more frequent. In addition, we want to avoid fate sharing for applications: changes to an application's datapath should not impact the performance of other applications. Ultimately, Snap is a network stack that does not contain application-specific code, where as mRPC needs to be application-aware for marshalling RPCs.

[4] For RDMA, if the number of disjoint memory blocks exceeds the limit of NIC's capability to encapsulate all blocks in one RDMA work request, mRPC coalesces the data into a memory block before transmission. This is because sending a single work request (even with a copy) is faster than sending multiple smaller work requests on our hardware.

We implement engines as plug-in modules that are dynamically loadable libraries. We design a live upgrade method that supports *upgrading, adding, or removing components of the datapath without disrupting other datapaths*.

Upgrading an engine. To upgrade one engine, mRPC first detaches the engine from its runtime (preventing it from being scheduled). Next, mRPC destroys and deallocates the old engine, but maintains the old engine's state in memory; note that the engine is detached from its queues and not running at this time. Afterwards, mRPC loads the new engine and configures its send and receive queues. The new engine starts with the old engine's state. If there is a change in the data structures of the engine's state, the upgraded engine is responsible for transforming the state as necessary (which the engine developer must implement). Note that this also applies to any shared state for cross-datapath engines. The last step is for mRPC to attach the new engine to the runtime.

Changing the datapath. When an operator changes the datapath to add or remove an engine, this process now involves the creation (or destruction) of queues and management of in-flight RPCs. Changes that add an engine are straightforward, since it only involves detaching and reconfiguring the queues between engines. Changes that remove an engine are more complex, as some in-flight RPCs may be maintained in internal buffers; for example, a rate limiter policy engine maintains an internal queue to ensure that the output queue meets a configured rate. Engine developers are responsible for flushing such internal buffers to the output queues when the engines are removed.

Multi-host upgrades or datapath changes. Some engine upgrades or datapath changes that involve both the sender and the receiver hosts need to carefully manage in-flight RPCs across hosts. For example, if we want to upgrade how mRPC uses RDMA, both the sender and the receiver have to be upgraded. In this scenario, the operator has to develop an upgrade plan that may involve upgrading an existing engine to some intermediate, backward-compatible engine implementation. The plan also needs to contain the upgrade sequence, e.g., upgrading the receiver side before the sender side. Our evaluation demonstrates such a complex live upgrade, which optimizes the handling of many small RPC requests over RDMA (see §7.3).

4.4 Security Considerations

We envision two deployment models for mRPC: (1) a cloud tenant uses mRPC to manage its RPC workloads (similar to how sidecars are used today); (2) a cloud provider uses mRPC to manage RPC workloads on behalf of tenants. In both models, there are two different classes of principals: operators and applications. Operators are responsible for configuring the hardware/virtual infrastructure, deploying the mRPC service, and setting up policies that mRPC will enforce. Applications run on an operator's infrastructure, interacting with the mRPC service to invoke RPCs. Applications trust operators, along with all privileged software (e.g., OS) and hardware that the operators provide; both applications and operators trust our mRPC service and protocol compiler. In both deployment models, applications are not trusted and may be malicious (e.g., attempt to circumvent network policies).

In the first deployment model, mRPC service runs on top of a virtualized network that is dedicated to the tenant. Running arbitrary policy and observability code inside the mRPC service cannot attack other tenants' traffic since inter-tenant isolation is provided by the cloud provider. In the second deployment model, our current prototype does not support running tenant-provided policy implementation inside mRPC service. How to safely integrate tenant-provided policy implementation and a cloud provider's own policy implementation is a future work.

From the application point of view, we want to ensure that **mRPC provides equivalent security guarantees as compared to today's RPC library and sidecar approach**, which we discuss in terms of: 1) dynamic binding and 2) policy enforcement. Our dynamic binding approach involves the generation, compilation, and runtime loading of a shared library for (un)marshalling application RPCs. Given that the compiled code is based on the application-provided RPC schema, this is a possible vector of attack. The mRPC schema compiler is trusted with a minimal interface: other than providing the RPC schema, applications have no control on the process of how the marshalling code is generated. We open source our implementation of the compiler so that it can be publicly reviewed.

As for all of our RPC processing logic, policies are enforced over RPCs by operating over their representations in shared memory control queues and data buffers. With a naive shared memory implementation, this introduces a vector of attack by exploiting a time-of-check to time-of-use (TOCTOU) attack; for instance, the application could modify the RPC message after policy enforcement but before the transport engine handles it. In mRPC, we address this by copying data into an mRPC-private heap prior to executing any policy that operates over the content of an RPC (as opposed to metadata such as the length). Similarly, received RPCs cannot be placed in shared memory until all policies have been enforced, since otherwise applications could see received RPCs before policies have a chance to drop (or modify) them. Shared memory regions are maintained by the mRPC service on a per-application basis to provide isolation.

5 Advanced Manageability Features

mRPC's architecture creates an opportunity for advanced manageability features such as cross-application RPC scheduling. In this section, we present two such features that we developed on our policy engine framework to demonstrate the broader utility of our RPC-as-a-managed-service architecture.

Feature 1: Global RPC QoS. mRPC allows centralized RPC scheduling of cross-application workloads based on a global view of current outstanding RPCs. For example, mRPC can enforce a policy that prioritizes RPCs with earliest deadlines [86] across applications to support latency SLO or prioritizes

latency-sensitive workloads [101]. One challenge here is that a naive implementation may attempt to apply the QoS policy for datapaths spread over multiple runtimes (i.e., execution thread contexts). This would require the (replicated) policy engines on each datapath to share the state on outstanding RPCs, and thus impose synchronization overheads. Therefore, we adopt a similar strategy as used in the Linux kernel to apply the QoS policy on a per-runtime basis, which instead can use runtime-local storage without the need for synchronization. In our implementation, we support a QoS strategy that prioritizes small RPCs based on a configurable threshold size.

Feature 2: Avoiding RDMA performance anomalies. It is well known that RDMA workloads may not fully utilize the capability of a specific RDMA NIC without fine-tuning, and that particular traffic patterns can even cause performance anomalies [40, 49] (e.g., low RDMA throughput, pause frame storms). Previous work such as ScaleRPC [13] and Flock [63] have proposed techniques to utilize the RNIC more efficiently. However, their approaches are library-based and only work for single applications; therefore, they do not handle scenarios in which the *combination* of multiple application workloads causes poor RDMA performance. mRPC's architecture enables us to have a global view of all RDMA requests and to avoid such performance anomalies.

We implement a global RDMA scheduler inside the RDMA transport engine, which translates RPC requests into RDMA messages and sends them to the RDMA NIC. In our implementation, we focus on addressing the performance degradation from interspersed small and large scatter-gather elements (which may be across RPCs as well as applications). We fuse such elements together with an explicit copy with an upper bound of 16 KB for the size of the fused element.

6 Implementation

mRPC is implemented in 32K lines of Rust: 3K lines for the protocol compiler, 6K for the mRPC control plane, 12K for engine implementations, and 11K for the mRPC library. The mRPC control plane is part of the mRPC service that loads/unloads engines.

The mRPC control plane is not live-upgradable. The mRPC library is linked into applications and is thus also not live-upgradable. We do not envision the need to frequently upgrade these components because they only implement the high-level, stable APIs, such as shared memory queue communication and (un)loading engines.

Engine interface. Table 1 presents the essential API functions that all engines must implement. Each engine represents some asynchronous computation that operates over input and output queues via `doWork`, which is similar in nature to Rust's `Future`. mRPC uses a pool of runtime executors to drive the engines by calling `doWork`, where each runtime executor corresponds to a kernel thread. We currently implement a simple scheduling strategy inspired by Snap [58]: engines can be scheduled to

Operations
`doWork`(in:[Queue], out:[Queue])
Operate over one or more RPCs available on input queues.
`decompose`(out:[Queue]) → State
Decompose the engine to its compositional states.
(Optionally output any buffered RPCs)
`restore`(State) → Engine
Restore the engine from the previously decomposed state.

Table 1: mRPC Engine Interface.

a dedicated or shared runtime on start. In addition, runtimes with no active engines will be put to slept and release CPU cycles. The engines also implement APIs to support live upgrading: `decompose` and `restore`. In `decompose`, the engine implementation is responsible for destructing the engine and creating a representation of the final state of the engine in memory, returning a reference to mRPC. mRPC invokes `restore` on the upgraded instance of the engine, passing in a reference to the final state of the old engine. The developer is responsible for handling backward compatibility across engine versions, similar to how application databases may be upgraded across changes to their schemas.

Transport engines. We abstract reliable network communication of messages into transport engines, which share similar design philosophy with Snap [58] and TAS [45]. We currently implement two transport engines: RDMA and TCP. Our RDMA transport engine is implemented based on OFED libibverbs 5.4, while our TCP transport engine is built on Linux kernel's TCP socket.

mRPC Library. Modern RPC libraries allow the user to specify the RPC data types and service interface through a language-independent schema file (e.g., `protobuf` for gRPC, `thrift` for Apache Thrift). mRPC implements support for `protobuf` and adopts similar service definitions as gRPC, except for gRPC's streaming API. mRPC also integrates with Rust's async/await ecosystem for ease of asynchronous programming in application development.

To create an RPC service, the developer only needs to implement the functions declared in the RPC schema. The dependent RPC data types are automatically generated and linked with the application by the mRPC schema compiler. The mRPC library handles all the rest, including task dispatching, thread management, and error handling. To allow applications to directly allocate data in shared memory without changing the programming abstraction, we implement a set of shared memory data structures that expose the same rich API as Rust's standard library. This is done by replacing the memory allocation of data structures such as `Vec` and `String` with the shared memory heap allocator.

7 Evaluation

We evaluate mRPC using an on-premise testbed of servers with two 100 Gbps Mellanox Connect-X5 RoCE NICs and two Intel 10-core Xeon Gold 5215 CPUs (running at 2.5 GHz

base frequency). The machines are connected via a 100 Gbps Mellanox SN2100 switch. Unless specified otherwise, we keep a single in-flight RPC to evaluate latency. To benchmark goodput and RPC rate, we let each client thread keep 128 concurrent RPCs on TCP and 32 concurrent RPCs on RDMA.

7.1 Microbenchmarks

We first evaluate mRPC's performance through a set of microbenchmarks over two machines, one for the client and the other for the server. The RPC request has a byte-array argument, and the response is also a byte array. We adjust the RPC size by changing the array length. RPC responses are an 8-byte array filled with random bytes. We compare mRPC with two state-of-the-art RPC implementations, eRPC and gRPC (v1.48.0). We deploy Envoy (v1.20) in HTTP mode to serve as a sidecar for gRPC. We use mRPC's TCP and RDMA backends to compare with gRPC and eRPC, respectively. There is no existing sidecar that supports RDMA. To evaluate the performance of using a sidecar to control eRPC traffic, we implement a single-thread sidecar proxy using the eRPC interface. We keep applications running for 15 seconds to measure the result.

Small RPC latency. We evaluate mRPC's latency by issuing 64-byte RPC requests over a single connection. Table 2 shows the latency for small RPC requests. Note that since the marshalling of small messages is fast on modern CPUs, the result in the table remains stable even when the message size scales up to 1 KB. We use `netperf` and `ib_read_lat` to measure raw round-trip latency.

mRPC achieves median latency of 32.8 µs for TCP and 7.6 µs for RDMA. Relative to netperf (TCP) or a raw RDMA read, mRPC adds 11.8 or 5.1 µs to the round-trip latency. This is the cost of the mRPC abstraction on top of the raw transport interface (e.g., socket, verbs).

We also evaluate latency in the presence of sidecar proxies. The sidecars do not enforce any policies, so we are only measuring the base overhead. Our results show that adding sidecars substantially increases the RPC latency. On gRPC, adding Envoy sidecars more than triples the median latency. The result is similar with eRPC. On mRPC, having a NullPolicy engine (which simply forwards RPCs) in the mRPC service has almost no effect on latency, increasing the median latency only by 300 ns.

Comparing the full solution (mRPC with policy versus gRPC/eRPC with proxy), mRPC speeds up the median latency by 6.1× (i.e., 33.4 µs against 203.4 µs) and the 99th percentile tail latency by 5.8×. On RDMA, mRPC speeds up eRPC by 1.3× and 1.4× in terms of median and tail latency (respectively). This is because the communication between the eRPC app and its proxy goes through the NIC, which triples the cost in the end-host driver (including the PCIe latency). In contrast, mRPC's architecture shortcuts this step with shared memory.

In addition, to separate the performance gain from system implementation difference, we evaluate the latency of mRPC with full gRPC-style marshalling (protobuf encoding and HTTP/2 framing) in the presence of NullPolicy engines as an ablation study. Under this setting, compared with gRPC + Envoy, mRPC speeds up the latency by 4.1× in terms of both median and tail latency. We also observe that the mRPC framework does not introduce significant overhead. Even with the cost of protobuf and HTTP/2 encoding, mRPC still achieves slightly lower latency compared with standalone gRPC. In mRPC, we can choose a customized marshalling format, because we know the other side is also an mRPC service. In other cases, e.g., when interfacing with external traffic or dealing with endianness differences, we can still apply full-gRPC style marshalling. When mRPC is configured to use full-gRPC style marshalling, we only need to pay (un)marshalling costs between mRPC services. For gRPC + Envoy, in addition to the (un)marshalling costs between Envoy proxies, the communication between applications and Envoy proxies also needs to pay this (un)marshalling cost. In the remaining evaluations, we will use mRPC's customized marshalling protocol. More results using gRPC-style marshalling are shown in §A.1.

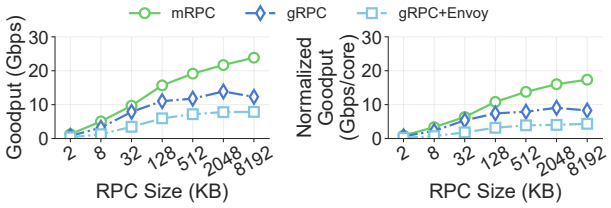
Transport	Solution	Median Latency (µs)	P99 Latency (µs)
TCP	Netperf	21.0	32.0
	gRPC	63.0	90.3
	mRPC	32.8	38.7
	gRPC+Envoy	203.4	251.1
	mRPC+NullPolicy	**33.4**	**43.3**
	mRPC+NullPolicy+HTTP+PB	49.8	61.9
RDMA	RDMA read	2.5	2.8
	eRPC	3.6	4.1
	mRPC	7.6	8.7
	eRPC+Proxy	11.3	15.6
	mRPC+NullPolicy	**7.9**	**9.1**

Table 2: **Microbenchmark [Small RPC latency]:** Round-trip RPC latencies for 64-byte requests and 8-byte responses.

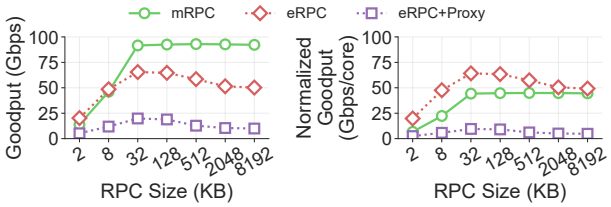
Large RPC goodput. The client and server in our goodput test use a single application thread. The left side of Figure 4 shows the result. From this point on, when we discuss mRPC's performance, we focus on the performance of mRPC that has at least a NullPolicy engine in place to fairly compare with sidecar-based approaches.

mRPC speeds up gRPC + Envoy and eRPC + Proxy, by 3.1× and 9.3×, respectively, for 8KB RPC requests. mRPC is especially efficient for large RPCs[5], for which (un)marshalling takes a higher fraction of CPU cycles in the end-to-end RPC datapath. Having a sidecar substantially hurts RPC goodput both for TCP and RDMA. In particular, for RDMA, intra-host roundtrip traffic through the RNIC might contend with inter-host traffic in the RNIC/PCIe bus, halving the available bandwidth for inter-host traffic. mRPC even outperforms gRPC (without Envoy). mRPC is fundamentally more efficient in terms of marshalling format: mRPC uses `iovec` and incurs no data movement. §A.1 shows an ablation study that demonstrates that even if mRPC uses a full gRPC-style marshalling engine, mRPC outperforms gRPC + Envoy due to a reduction in the number of (un)marshalling steps.

CPU overheads. To understand the mRPC CPU overheads, we measure the per-core goodput. The results are shown on

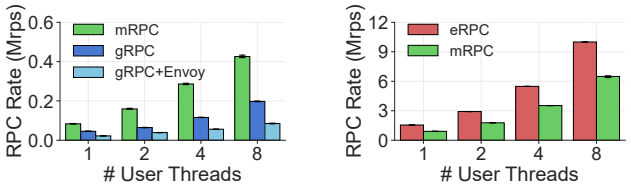


(a) TCP-based transport



(b) RDMA-based transport

Figure 4: **Microbenchmark [Large RPC goodput]:** Comparison of goodput for large RPCs. Note that different solutions demand different amounts of CPU cores, so we also normalized the goodput to their CPU utilization, as shown in the right figures. The error bars show the 95% confidence interval, but they are too small to be visible.



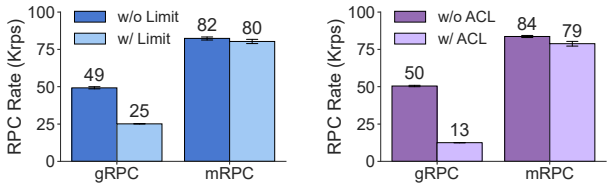
(a) TCP-based transport (b) RDMA-based transport

Figure 5: **Microbenchmark [RPC rate and scalability]:** Comparison of small RPC rate and CPU scalability. The bars show the RPC rate. The error bars show the 95% confidence interval.

the right side of Figure 4. mRPC speeds up gRPC + Envoy and eRPC + Proxy, by 3.8× and 9.3×, respectively. This means mRPC is much more CPU-efficient than gRPC + Envoy and eRPC + Proxy. eRPC (without a proxy) is quite efficient, but converges to mRPC's efficiency as RPC size increases.

RPC rate and scalability. We evaluate mRPC's small RPC rate and its multicore scalability. We fix the RPC request size to 32 bytes and scale the number of client threads. We use the same number of threads for the server as the client, and each client connects to one server thread. Figure 5 shows the RPC rates when scaling from 1 to 8 user threads. All the tested solutions scale well. mRPC's RPC rates scale by 5.1× and 7.2×, on TCP and RDMA, from a single thread to 8 threads. As a reference, gRPC scales by 4.3×, gRPC + Envoy scales by 3.9×, and eRPC scales by 6.5×. mRPC achieves 0.43 Mrps on TCP and 6.5 Mrps on RDMA with 8 threads. gRPC + Envoy only has 0.09 Mrps, so mRPC outperforms it by 5×. We do not evaluate eRPC + proxy, because our eRPC proxy is only single-threaded. When we run eRPC + proxy with a single thread, it achieves 0.51 Mrps. So even if eRPC + proxy scales linearly to 8 threads, mRPC still outperforms it.

[5]Standalone eRPC exhibits relatively lower goodput on RoCE than on Infiniband. According to the eRPC paper [39], eRPC should achieve 75 Gbps on Infiniband for 8MB RPCs.



(a) Rate Limiting (b) Access Control

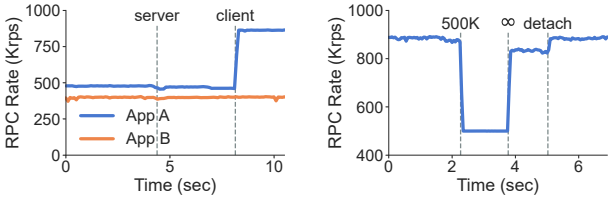
Figure 6: **Efficient Support for Network Policies**. The RPC rates with and without policy are compared. The bars of w/o Limit and w/o ACL for gRPC show its throughput when the sidecar is bypassed. The error bars show the 95% confidence interface.

7.2 Efficient Policy Enforcement

We use two network policies as examples to demonstrate mRPC's efficient support for RPC policies: (1) RPC rate limiting and (2) access control based on RPC arguments. RPC rate limiting allows an operator to specify how many RPCs a client can send per second. We implement rate limiting as an engine using the token bucket algorithm [91]. Our access control policy inspects RPC arguments and drops RPCs based on a set of rules specified by network operators. These two network policies differ greatly from traditional rate limiting and access control, which only limit network bandwidth and can only operate on packet headers.

We compare rate limit enforcement using an mRPC policy versus using Envoy's rate limiter on gRPC workloads. To evaluate the performance overheads, we set the limit to infinity so that the actual RPC rate is never above the limit (allowing us to observe the overheads). Figure 6a shows the RPC rate with and without the rate limits. gRPC's RPC rate drops immediately from 49K to 25K. This is because having a sidecar proxy (Envoy) introduces substantial performance overheads. For mRPC, the RPC rate stays the same at 82K. This is because having a policy introduces minimal overheads. The extra policy only adds tens to hundreds of extra CPU instructions on the RPC datapath.

We evaluate access control on a hotel reservation application in DeathStarBench [23]. The service handles hotel reservation RPC requests, which include the customer's name, the check-in date, and other arguments. The service then returns a list of recommended hotel names. We set the access control policy to filter RPCs based on the `customerName` argument in the request. We use a synthetic workload containing 99% valid and 1% invalid requests. We again compare our mRPC policy



(a) Transport adapter (b) Rate limit policy

Figure 7: **Live upgrade**. In (a), the annotations indicate when the client of App A and server of A and B are upgraded. In (b), the annotations denote the specified rate and when the policy is removed.

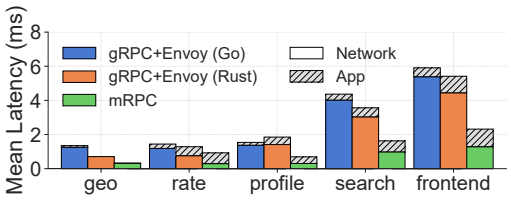


Figure 8: **DeathStarBench**: Mean latency of in-application processing and network processing of microservices. The latency of a microservice includes RPC calls to other microservices. The frontend latency represents complete end-to-end latency.

against using Envoy to filter gRPC requests. We implement the Envoy policy using WebAssembly. gRPC's rate drops from 50K to 13K. This is because of the same sidecar overheads and now Envoy has to further parse the packets to fetch the RPC arguments. On mRPC, the performance drop is much smaller, from 84K to 79K. Note that, on mRPC, the performance overhead of introducing access control is larger than rate limiting. For access control, the mRPC service has to copy the relevant field (i.e., `customerName`) to the private heap to prevent TOCTOU attacks on the sender side and has to copy the RPC from a private heap to the shared heap on the receiver side.

7.3 Live Upgrade

We demonstrate mRPC's ability to live upgrade using two scenarios.

Scenario 1. During our development of mRPC, we realized that using the RDMA NIC's scatter-gather list to send multiple arguments in a single RPC can significantly boost mRPC's performance. In this approach, even when an RPC contains arguments that are scattered in virtual memory, we can send the RPC using a single RDMA operation (`ibv_post_send`). We use these two versions of our RDMA transport engine to demonstrate that mRPC enables such an upgrade without affecting running applications. Note that all other evaluations already include this RDMA feature. This upgrade involves both the client side's mRPC service and the server side's mRPC service, because it involves how RDMA is used between machines (i.e., transport adapter engine). gRPC and eRPC cannot support this type of live upgrade.

We run two applications (App A and App B). Both applications are sending 32-byte RPCs, and the responses are 8 bytes. A and B share the mRPC service on the server side. A's and B's RPC clients are on different machines. We keep 8 concurrent RPCs for B, forcing it to send at a slower rate, while using 32 for A. We first upgrade the server side to accept arguments as a scatter-gather list, and we then upgrade the client side of A. Figure 7a shows the RPC rate of A and B. When the server side upgrades, we observe a negligible effect on A's and B's rate. Neither A nor B needs recompilation or rebooting. When A's client side's mRPC service is upgraded, A's performance increases from 480K to 860K. B's performance is not affected at all because B's client side's mRPC service is not upgraded.

Scenario 2. Enforcing network policies has performance overheads, even when they do not have any effect. For example, enforcing a rate limit of an extremely large throttle rate still introduces performance overheads just for tracking the current rate using token buckets. mRPC allows policies to be removed at runtime, without disrupting running applications.

We use the same rate limiting setup from §7.2 but on top of RDMA transport. Figure 7b shows the RPC rate. We start from not having the rate limit engine. We then load the rate limit engine and set the throttled rate to 500K. The RPC rate immediately becomes 500K. We then set the throttled rate to be infinite, and the rate becomes 840K. After we detach the rate limit engine, the rate becomes 890K.

Takeaways. There are two overall takeaways from these experiments. First, mRPC allows upgrades to the mRPC service without disrupting running applications. Second, live upgrades allow for more flexible management of RPC services, which can be used to enable immediate performance improvements (without redeploying applications) or dynamic configuration of policies.

7.4 Real Applications

We evaluate how the performance benefits of mRPC transform into end-to-end application-level performance metrics.

DeathStarBench. We use the hotel reservation service from the DeathStarBench [23] microservice benchmark suite. The reference benchmark is implemented in Go with gRPC and Consul [15] (for service discovery). Our mRPC prototype currently only supports Rust applications, and we thus port the application code to Rust for comparison. We use the same open-source services such as memcached [59] and MongoDB [64].

We distribute the HTTP frontend and the microservices on four servers in our testbed. The monolithic services (memcached, MongoDB) are co-located with the microservices that depend on them. We use a single thread for each of the microservices and the frontend. Further, we deploy an Envoy proxy as a sidecar on each of the servers (with no active policy). The pro-

	Median Latency	P99 Latency	Throughput
eRPC	16.8 µs	21.7 µs	8.7 MOPS
mRPC	22.5 µs	33.1 µs	7.0 MOPS

Table 3: **Masstree analytics**: Latency and the achieved throughput for GET operations. MOPS is Million Operations Per Second.

vided workload generator [23] is used to submit HTTP requests to the frontend. For a fair comparison, we also implemented a Rust version of the benchmark with Tonic [93], which is the de facto implementation of gRPC in Rust. We deploy the mRPC and Tonic implementations on bare metal, while the reference Go suite runs in Docker containers with a host network (which introduces negligible performance overheads compared to using bare metal [103]). All three solutions are based on TCP. We issue 20 requests per second for 250 seconds and record the latency of each request, breaking it down into the in-application processing time and network processing time for each microservice involved. In our evaluation, the dynamic bindings of the user applications are already cached in mRPC service, so the time to generate the bindings is not included in the result.

Figure 8 shows the latency breakdown. First, we validate that our own implementation of DeathStarBench on Rust is a faithful re-implementation. We can see that the original Go implementation and our Rust implementation have similar latency. Moreover, the amount of latency spent in gRPC is similar. Second, mRPC with a null policy outperforms by 2.5× gRPC with a sidecar proxy in average end-to-end latency. §A.2 contains more details about the tail latency and the scenario without a sidecar.

Masstree analytics. We also evaluate the performance of Masstree [56], an in-memory key-value store, over both mRPC and eRPC [39] using RDMA. We follow the exact same workload setup used in eRPC, which contains 99% I/O-bounded point GET request and 1% CPU-bounded range SCAN request. We run the Masstree server on one machine and run the client on another machine. Both the server and the client use 10 threads, with each client thread using 16 concurrent requests. The test runs for 60 seconds. The result in Table 3 shows that eRPC outperforms mRPC, which makes sense since eRPC is a well-designed library implementation that is focused on high performance. mRPC enables many other manageability features in exchange for a slight reduction in performance. In this case, using mRPC instead of eRPC means that median latency increases by 34% and throughput reduces by 20%.

7.5 Benefits of Advanced Manageability Features

Next, we demonstrate the performance benefits of having centralized RPC management, through two advanced manageability features that we developed (see §5). We use synthetic workloads to test the advanced manageability features.

	Latency App		B/W App
	P95 Latency	P99 Latency	Bandwidth
w/o QoS	45.1 µs	54.6 µs	22.2 Gbps
w/ QoS	19.5 µs	21.8 µs	22.0 Gbps

Table 4: **Global QoS**: Performance of latency- and bandwidth-sensitive applications with and without a global QoS policy.

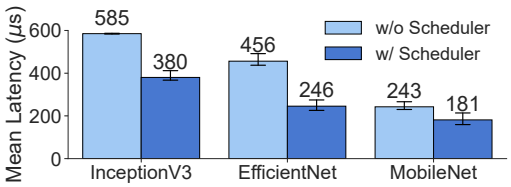


Figure 9: **RDMA Scheduler**: Mean RPC latency with or without RDMA scheduler. The error bars show the 95% confidence interval.

Global RPC QoS. We enable our cross-application QoS policy that reorders requests from multiple applications and prioritizes small RPC quests. We set up two applications and pin them to the same mRPC runtime. One application is latency-sensitive, sending 32-byte RPC requests with a single RPC in-flight; the other is bandwidth-sensitive, sending 32 KB requests with 64 concurrent RPCs. We measure the tail latency for the latency-sensitive application and the utilized bandwidth of the bandwidth-sensitive one.

Table 4 shows the result. Without the QoS policy, the bandwidth-sensitive application has a high bandwidth utilization; however, the latency-sensitive application suffers from a high tail latency. With the QoS policy in place, the small requests from the latency-sensitive application get higher priority and are sent first, improving P99 latency from 54.6 µs to 21.8 µs. Since small RPC requests consume negligible bandwidth, it barely affects the bandwidth-sensitive application (less than a 1% bandwidth drop).

RDMA Scheduler. Our RDMA scheduler batches small RPC requests into (at most) 16KB messages and sends requests using a single RMDA operation to reduce the load on the RDMA NIC. Our synthetic workload is based on BytePS [37], which uses RDMA for distributed deep learning. To synchronize a tensor to/from a server, BytePS prepends an 8-byte key and appends a 4-byte length to describe the tensor. The three disjoint memory blocks are placed in a scatter-gather list and submitted to the NIC, resulting in a small-large-small message pattern that triggers a performance anomaly [49]. This message pattern is quite common in real applications, as programs often need to describe a large payload with a small piece of metadata. We emulate BytePS's RPC request pattern and generate RPCs from three widely-used models: MobileNet, EfficientNetB0, and InceptionV3 [31, 89, 90]. Each RPC call consists of an

8-byte key, a payload of tensor, and a 4-byte length. We use a single thread to make RPCs. Figure 9 shows the average RPC latency. The RDMA scheduler provides 30-90% latency improvement. This improvement differs for different neural networks, because of different RDMA message patterns.

8 Related Work

Fast RPC implementations. Optimizing RPC has a long history. Birrell and Nelson's early RPC design [10] includes generating bindings via a compiler, interfacing with transport protocols, and various optimizations (e.g., implicit ACK). Bershad et al. showed how to use shared-memory queues to efficiently pass RPC messages between processes on the same machine [8]. mRPC's shared-memory region leverages this idea but extends it to allow for marshalling code to be applied after policy enforcement. A similar use of shared-memory queues can be found with recent Linux support for asynchronous system calls [3] combined with scatter-gather I/O [54]; unlike traditional system calls, however, mRPC protocol descriptions can be defined at runtime.

Another line of work uses RDMA to speed network RPCs [13, 39, 41, 63, 87, 88]. These studies assume direct application access to network hardware and are thus susceptible to RDMA's security weaknesses [79]. mRPC leverages ideas from RDMA RPC research but in a way that is compatible with policy enforcement and observability, by doing so as a service. Another line of work reduces the cost of marshalling, by using alternative formats [2, 9, 11, 20, 38, 66, 78, 92] or designing hardware accelerators [35, 43, 76, 97]. This work is largely orthogonal to our goal of removing unnecessary marshalling steps but could be applied to further improve mRPC performance.

Fast network stacks. Building efficient host network stacks is a popular research topic. MegaPipe [28], mTCP [36], Arrakis [73], IX [5], eRPC [39], and Demikernel [99] advocate building the network stack as a user-level library, bypassing the kernel for performance. In these systems, an application directly accesses the network interface, but they assume policy can be enforced by the network hardware and are thus vulnerable if the hardware has security weaknesses. mRPC can interpose policy on any RPC. Like mRPC, Snap [58] and TAS [45] implement the network stack as a service, but they stop at layer 4 (TCP and UDP) rather than layer 7 (RPC). Application RPC stubs must marshal data into shared memory queues to use Snap or TAS. Flexible policy engines are a key feature of Snap, but because Snap operates at layer 4, it can only apply layer 7 policies by unmarshalling and re-marshalling RPC data. A fast network stack like mRPC can also be implemented directly in the kernel. LITE [95] implements RDMA operations as system calls inside the kernel to improve manageability, and Shenango [69] interposes a specialized kernel packet scheduler for network messages.

Fast network proxies. There is a long line of work on improving the performance of network proxies [33, 34, 44, 46, 47, 51, 57, 60, 70, 71, 74, 75, 85, 100]. Much of this work considers the general case of a standalone proxy. Our work differs in two ways. First, our proposed technique is only for RPC traffic rather than generalized TCP traffic. Second, we co-design the application library stub and proxy, and thus, both must be co-located on the same machine for our shared memory queues to function. In today's sidecar proxies (our baseline), this assumption holds, but it does not hold for generalized network proxies.

Live upgrades of system software. Being able to update system software without disrupting or restarting applications is key to achieving end-to-end high availability. Snap [58] provides live upgrade of the network stack running as a proxy; Bento [61] provides similar functionality for kernel-resident file systems. Relative to these systems, mRPC upgrades are more fine-grained. For example, Snap targets a maximum outage during upgrades of 200 milliseconds, by spawning another instance of itself and moving all connections to the new process. By contrast, our goal is near instantaneous changes and upgrades to RPC protocol definitions, policy engines, and marshalling code. We accomplish this by keeping the control plane intact and performing updates by loading and unloading dynamic libraries. eBPF is a Linux kernel extensibility mechanism that supports dynamic updates [17]; unlike eBPF, mRPC can dynamically change the execution graph of policy engines as well as the individual engines themselves.

9 Conclusion

Remote procedure call has become the de facto abstraction for building distributed applications in datacenters. The increasing demand for manageability makes today's RPC libraries inadequate. Inserting a sidecar proxy into the network datapath allows for manageability but slows down RPC substantially due to redundant marshalling and unmarshalling. We present mRPC, a novel architecture to implement RPC as a managed service to achieve both high performance and manageability. mRPC eliminates the redundant marshalling overhead by applying policy to RPC data before marshalling and only copying data when necessary for security. This new architecture enables live upgrade of RPC processing logic and new RPC scheduling and transport methods to improve performance. We have performed extensive evaluations through a set of micro-benchmarks and two real applications to demonstrate that mRPC enables a unique combination of high performance, policy flexibility, security, and application-level availability. Our source code is available at `https://github.com/phoenix-dataplane/phoenix`.

Acknowledgement

We thank our shepherd Amy Ousterhout and other anonymous reviewers for their insightful feedback. Our work is partially supported by NSF grant CNS-2213387 and by gifts from Adobe, Amazon, IBM, and Meta.

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: A System for Large-Scale Machine Learning. In *OSDI*, 2016.

[2] Apache Arrow. https://arrow.apache.org/, 2022.

[3] Jens Axboe. Efficient IO with io_uring. https://kernel.dk/io_uring.pdf, 2019.

[4] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The Multikernel: A New OS Architecture for Scalable Multicore Systems. In *SOSP*, 2009.

[5] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In *OSDI*, 2014.

[6] Benjamin Berg, Daniel S. Berger, Sara McAllister, Isaac Grosof, Sathya Gunasekar, Jimmy Lu, Michael Uhlar, Jim Carrig, Nathan Beckmann, Mor Harchol-Balter, and Gregory R. Ganger. The CacheLib Caching Engine: Design and Experiences at Scale. In *OSDI*, 2020.

[7] Brian N. Bershad, Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy. Lightweight Remote Procedure Call. *ACM Trans. Comput. Syst.*, 8(1):37–55, February 1990.

[8] Brian N. Bershad, Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy. User-Level Interprocess Communication for Shared Memory Multiprocessors. *ACM Trans. Comput. Syst.*, 1991.

[9] Bincode. https://github.com/bincode-org/bincode, 2022.

[10] Andrew D. Birrell and Bruce Jay Nelson. Implementing Remote Procedure Calls. *ACM Trans. Comput. Syst.*, 1984.

[11] Cap'n Proto. https://capnproto.org/, 2022.

[12] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache Flink: Stream and Batch Processing in a Single Engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 36(4), 2015.

[13] Youmin Chen, Youyou Lu, and Jiwu Shu. Scalable RDMA RPC on Reliable Connection with Efficient Resource Sharing. In *EuroSys*, 2019.

[14] Inho Cho, Ahmed Saeed, Joshua Fried, Seo Jin Park, Mohammad Alizadeh, and Adam Belay. Overload Control for µs-scale RPCs with Breakwater. In *OSDI*, 2020.

[15] Consul. https://www.consul.io/, 2022.

[16] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*, 2004.

[17] eBPF. https://ebpf.io/, 2022.

[18] Envoy Proxy. https://www.envoyproxy.io/, 2022.

[19] etcd. https://etcd.io/, 2022.

[20] FlatBuffers. https://google.github.io/flatbuffers/, 2022.

[21] Jason Flinn, Xianzheng Dou, Arushi Aggarwal, Alex Boyko, Francois Richard, Eric Sun, Wendy Tobagus, Nick Wolchko, and Fang Zhou. Owl: Scale and Flexibility in Distribution of Hot Content. In *OSDI*, 2022.

[22] Rodrigo Fonseca, George Porter, Randy H. Katz, and Scott Shenker. X-Trace: A Pervasive Network Tracing Framework. In *NSDI*, 2007.

[23] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Yuan He, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, and Christina Delimitrou. An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems. In *ASPLOS*, 2019.

[24] Gluster. https://www.gluster.org/, 2022.

[25] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In *OSDI*, 2012.

[26] gRPC. https://grpc.io/, 2022.

[27] gRPC Release Schedule. https://grpc.github.io/grpc/core/md_doc_grpc_release_schedule.html, 2022.

[28] Sangjin Han, Scott Marshall, Byung-Gon Chun, and Sylvia Ratnasamy. MegaPipe: A New Programming Interface for Scalable Network I/O . In *OSDI*, 2012.

[29] HAProxy. http://www.haproxy.org/, 2022.

[30] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. In *NSDI*, 2011.

[31] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. https://arxiv.org/abs/1704.04861, 2017.

[32] Istio. https://istio.io/, 2022.

[33] Ethan J. Jackson, Melvin Walls, Aurojit Panda, Justin Pettit, Ben Pfaff, Jarno Rajahalme, Teemu Koponen, and Scott Shenker. SoftFlow: A Middlebox Architecture for Open vSwitch. In *ATC*, 2016.

[34] Muhammad Asim Jamshed, YoungGyoun Moon, Donghwi Kim, Dongsu Han, and KyoungSoo Park. mOS: A Reusable Networking Stack for Flow Monitoring Middleboxes. In *NSDI*, 2017.

[35] Jaeyoung Jang, Sung Jun Jung, Sunmin Jeong, Jun Heo, Hoon Shin, Tae Jun Ham, and Jae W Lee. A Specialized Architecture for Object Serialization with Applications to Big Data Analytics. In *ISCA*, 2020.

[36] EunYoung Jeong, Shinae Wood, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. mTCP: a Highly Scalable User-level TCP Stack for Multicore Systems. In *NSDI*, 2014.

[37] Yimin Jiang, Yibo Zhu, Chang Lan, Bairen Yi, Yong Cui, and Chuanxiong Guo. A Unified Architecture for Accelerating Distributed DNN Training in Heterogeneous GPU/CPU Clusters. In *OSDI*, 2020.

[38] Introducing JSON. https://www.json.org/json-en.html, 2022.

[39] Anuj Kalia, Michael Kaminsky, and David Andersen. Datacenter RPCs can be General and Fast. In *NSDI*, 2019.

[40] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Design Guidelines for High Performance RDMA Systems. In *USENIX ATC*, 2016.

[41] Anuj Kalia, Michael Kaminsky, and David G. Andersen. FaSST: Fast, Scalable and Simple Distributed Transactions with Two-Sided (RDMA) Datagram RPCs. In *OSDI*, 2016.

[42] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. Profiling a Warehouse-Scale Computer. In *ISCA*, 2015.

[43] Sagar Karandikar, Chris Leary, Chris Kennelly, Jerry Zhao, Dinesh Parimi, Borivoje Nikolic, Krste Asanovic, and Parthasarathy Ranganathan. A Hardware Accelerator for Protocol Buffers. In *MICRO*, 2021.

[44] Georgios P. Katsikas, Tom Barbette, Dejan Kostić, Rebecca Steinert, and Gerald Q. Maguire Jr. Metron: NFV Service Chains at the True Speed of the Underlying Hardware. In *NSDI*, 2018.

[45] Antoine Kaufmann, Tim Stamler, Simon Peter, Naveen Kr Sharma, Arvind Krishnamurthy, and Thomas Anderson. TAS: TCP Acceleration as an OS Service. In *EuroSys*, 2019.

[46] Joongi Kim, Keon Jang, Keunhong Lee, Sangwook Ma, Junhyun Shim, and Sue Moon. NBA (Network Balancing Act): A High-Performance Packet Processing Framework for Heterogeneous Processors. In *EuroSys*, 2015.

[47] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The Click Modular Router. *ACM Trans. Comput. Syst.*, 2000.

[48] Xinhao Kong, Jingrong Chen, Wei Bai, Xu Yechen, Mahmoud Elhaddad, Shachar Raindel, Jitendra Padhye, Alvin R Lebeck, and Danyang Zhuo. Understanding RDMA Microarchitecture Resources for Performance Isolation. In *NSDI*, 2023.

[49] Xinhao Kong, Yibo Zhu, Huaping Zhou, Zhuo Jiang, Jianxi Ye, Chuanxiong Guo, and Danyang Zhuo. Collie: Finding Performance Anomalies in RDMA Subsystems. In *NSDI*, 2022.

[50] Kubernetes. https://kubernetes.io/, 2022.

[51] Bojie Li, Kun Tan, Layong (Larry) Luo, Yanqing Peng, Renqian Luo, Ningyi Xu, Yongqiang Xiong, Peng Cheng, and Enhong Chen. ClickNP: Highly Flexible and High Performance Network Processing with Reconfigurable Hardware. In *SIGCOMM*, 2016.

[52] Tianxi Li, Haiyang Shi, and Xiaoyi Lu. HatRPC: Hint-Accelerated Thrift RPC over RDMA. In *SC*, 2021.

[53] Linkerd. https://linkerd.io/, 2022.

[54] Rober Love. *Linux System Programming*. O'Reilly Media, 2007.

[55] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M. Hellerstein. Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud. *Proc. VLDB Endow.*, 5(8):716–727, 2012.

[56] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. Cache Craftiness for Fast Multicore Key-Value Storage. In *EuroSys*, 2012.

[57] Joao Martins, Mohamed Ahmed, Costin Raiciu, Vladimir Olteanu, Michio Honda, Roberto Bifulco, and Felipe Huici. ClickOS and the Art of Network Function Virtualization. In *NSDI*, 2014.

[58] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C. Evans, Steve Gribble, Nicholas Kidd, Roman Kononov, Gautam Kumar, Carl Mauer, Emily Musick, Lena Olson, Erik Rubow, Michael Ryan, Kevin Springborn, Paul Turner, Valas Valancius, Xi Wang, and Amin Vahdat. Snap: A Microkernel Approach to Host Networking. In *SOSP*, 2019.

[59] Memcached. https://memcached.org/, 2022.

[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 *SIGCOMM*, 2017.

[61] Samantha Miller, Kaiyuan Zhang, Mengqi Chen, Ryan Jennings, Ang Chen, Danyang Zhuo, and Thomas Anderson. High Velocity Kernel File Systems with Bento. In *FAST*, 2021.

[62] Jeffrey C. Mogul and John Wilkes. Nines Are Not Enough: Meaningful Metrics for Clouds. In *HotOS*, 2019.

[63] Sumit Kumar Monga, Sanidhya Kashyap, and Changwoo Min. Birds of a Feather Flock Together: Scaling RDMA RPCs with Flock. In *SOSP*, 2021.

[64] MongoDB. https://www.mongodb.com, 2022.

[65] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. Ray: A Distributed Framework for Emerging AI Applications. In *OSDI*, 2018.

[66] MessagePack. https://msgpack.org/index.html, 2022.

[67] Nginx. https://www.nginx.com/, 2022.

[68] Diego Ongaro and John Ousterhout. In Search of an Understandable Consensus Algorithm. In *ATC*, 2014.

[69] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads. In *NSDI*, 2019.

[70] Shoumik Palkar, Chang Lan, Sangjin Han, Keon Jang, Aurojit Panda, Sylvia Ratnasamy, Luigi Rizzo, and Scott Shenker. E2: A Framework for NFV Applications. In *SOSP*, 2015.

[71] Aurojit Panda, Sangjin Han, Keon Jang, Melvin Walls, Sylvia Ratnasamy, and Scott Shenker. NetBricks: Taking the V out of NFV. In *OSDI*, 2016.

[72] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *NeurIPS*. 2019.

[73] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. Arrakis: The Operating System is the Control Plane. In *OSDI*, 2014.

[74] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Joe Stringer, Pravin Shelar, Keith Amidon, and Martin Casado. The Design and Implementation of Open vSwitch. In *NSDI*, 2015.

[75] Salvatore Pontarelli, Roberto Bifulco, Marco Bonola, Carmelo Cascone, Marco Spaziani, Valerio Bruschi, Davide Sanvito, Giuseppe Siracusano, Antonio Capone, Michio Honda, Felipe Huici, and Giuseppe Siracusano. FlowBlaze: Stateful Packet Processing in Hardware. In *NSDI*, 2019.

[76] Arash Pourhabibi, Siddharth Gupta, Hussein Kassir, Mark Sutherland, Zilu Tian, Mario Paulo Drumond, Babak Falsafi, and Christoph Koch. Optimus Prime: Accelerating Data Transformation in Servers. In *ASPLOS*, 2020.

[77] Leon Poutievski, Omid Mashayekhi, Joon Ong, Arjun Singh, Mukarram Tariq, Rui Wang, Jianan Zhang, Virginia Beauregard, Patrick Conner, Steve Gribble, Rishi Kapoor, Stephen Kratzer, Nanfang Li, Hong Liu, Karthik Nagaraj, Jason Ornstein, Samir Sawhney, Ryohei Urata, Lorenzo Vicisano, Kevin Yasumura, Shidong Zhang, Junlan Zhou, and Amin Vahdat. Jupiter Evolving: Transforming Google's Datacenter Network via Optical Circuit Switches and Software-Defined Networking. In *SIGCOMM*, 2022.

[78] Protocol Buffers. https://developers.google.com/protocol-buffers, 2022.

[79] Benjamin Rothenberger, Konstantin Taranov, Adrian Perrig, and Torsten Hoefler. ReDMArk: Bypassing RDMA security mechanisms. In *USENIX Security*, 2021.

[80] Russel Sandberg. The Sun Network File System: Design, Implementation and Experience. In *USENIX Summer ATC*, 1986.

[81] Mike Schroeder and Michael Burrows. Performance of Firefly RPC. *ACM Transaction on Computer Systems*, February 1990.

[82] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The Hadoop Distributed File System. In *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, 2010.

[83] Arjun Singhvi, Aditya Akella, Maggie Anderson, Rob Cauble, Harshad Deshmukh, Dan Gibson, Milo M. K. Martin, Amanda Strominger, Thomas F. Wenisch, and Amin Vahdat. CliqueMap: Productionizing an RMA-Based Distributed Caching System. In *SIGCOMM*, 2021.

[84] Mark Slee, Aditya Agarwal, and Marc Kwiatkowski. Thrift: Scalable Cross-Language Services Implementation. *Facebook white paper*, 5(8):127, 2007.

[85] Snabb: Simple and fast packet networking. https://github.com/snabbco/snabb, 2022.

[86] Marco Spuri and Giorgio C. Buttazzo. Efficient aperiodic service under earliest deadline scheduling. In *Real-Time Systems Symposium*, pages 2–11, 1994.

[87] Patrick Stuedi, Animesh Trivedi, Bernard Metzler, and Jonas Pfefferle. DaRPC: Data Center RPC. In *SoCC*, 2014.

[88] Maomeng Su, Mingxing Zhang, Kang Chen, Zhenyu Guo, and Yongwei Wu. RFP: When RPC is Faster than Server-Bypass with RDMA. In *EuroSys*, 2017.

[89] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. Rethinking the Inception Architecture for Computer Vision. In *CVPR*, 2016.

[90] Mingxing Tan and Quoc Le. Efficientnet: Rethinking Model Scaling for Convolutional Neural Networks. In *ICML*, 2019.

[91] Puqi Perry Tang and Tsung-Yuan Charles Tai. Network Traffic Characterization Using Token Bucket Model. In *INFOCOM*, 1999.

[92] Apache Thrift. https://thrift.apache.org/, 2022.

[93] Tonic. https://github.com/hyperium/tonic, 2022.

[94] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M. Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, Nikunj Bhagat, Sailesh Mittal, and Dmitriy Ryaboy. Storm@twitter. In *SIGMOD*, 2014.

[95] Shin-Yeh Tsai and Yiying Zhang. LITE Kernel RDMA Support for Datacenter Applications. In *SOSP*, 2017.

[96] Amin Vadhat. Coming of Age in the Fifth Epoch of Distributed Computing: The Power of Sustained Exponential Growth, 2020. Amin Vahdat - SIGCOMM Lifetime Achievement Award 2020 Keynote.

[97] Adam Wolnikowski, Stephen Ibanez, Jonathan Stone, Changhoon Kim, Rajit Manohar, and Robert Soulé. Zerializer: Towards Zero-Copy Serialization. In *HotOS*, 2021.

[98] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *NSDI*, 2012.

[99] Irene Zhang, Amanda Raybuck, Pratyush Patel, Kirk Olynyk, Jacob Nelson, Omar S Navarro Leija, Ashlie Martinez, Jing Liu, Anna Kornfeld Simpson, Sujay Jayakar, et al. The Demikernel Datapath OS Architecture for Microsecond-scale Datacenter Systems. In *SOSP*, 2021.

[100] Kaiyuan Zhang, Danyang Zhuo, and Arvind Krishnamurthy. Gallium: Automated Software Middlebox Offloading to Programmable Switches. In *SIGCOMM*, 2020.

[101] Yiwen Zhang, Yue Tan, Brent Stephens, and Mosharaf Chowdhury. Justitia: Software Multi-Tenancy in Hardware Kernel-Bypass Networks. In *NSDI*, 2022.

[102] Xiangfeng Zhu, Guozhen She, Bowen Xue, Yu Zhang, Yongsu Zhang, Xuan Kelvin Zou, Xiongchun Duan, Peng-Ju He, Arvind Krishnamurthy, Matthew Lentz, Danyang Zhuo, and Ratul Mahajan. Dissecting Service Mesh Overheads. *ArXiv*, abs/2207.00592, 2022.

[103] Danyang Zhuo, Kaiyuan Zhang, Yibo Zhu, Hongqiang Harry Liu, Matthew Rockett, Arvind Krishnamurthy, and Thomas Anderson. Slim: OS Kernel Support for a Low-Overhead Container Overlay Network. In *NSDI*, 2019.

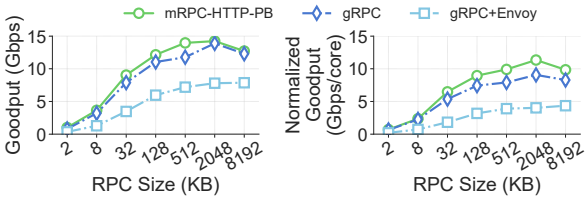


Figure 10: **Microbenchmark [Large RPC bandwidth]:** Comparison of large RPC bandwidth where we use HTTP/2 and protobuf (PB) marshalling for mRPC, on TCP transport. The error bars show the 95% confidence interval, but they are too small to be visible.

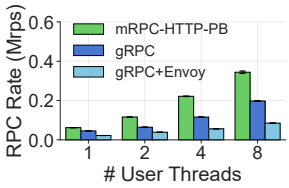


Figure 11: **Microbenchmark [RPC rate and scalability]:** Comparison of small RPC rate and CPU scalability where we use HTTP/2 and protobuf (PB) marshalling for mRPC, on TCP transport. The error bars show the 95% confidence interval.

A Appendix

A.1 mRPC with Full gRPC-style Marshalling

As gRPC uses `protobuf` [78] for encoding and HTTP/2 as the payload carrier, it has a memory copying and HTTP/2 framing cost. On the other hand, mRPC is agnostic to the marshalling format. Although mRPC's default marshalling is zero-copy and is generally faster than gRPC-style marshalling, our main goal of the paper is to show that we can eliminate the redundant (un)marshalling steps while enabling network policies and observability for RPC traffic.

To isolate the performance benefits of using zero-copy marshalling and reducing the number of (un)marshalling steps, we evaluate mRPC with full gRPC-style marshalling (protobuf + HTTP/2). We implement an mRPC variant that applies encoding (decoding) code generated by the `protobuf` compiler and HTTP/2 framing for inter-host mRPC service communication.

We conduct the same large RPC goodput experiment in §7.1. The results are presented in Figure 10. We find that mRPC achieves performance comparable to gRPC after switching to using protobuf + HTTP/2. With full gRPC marshalling, mRPC still performs 2.6× and 3.7× as fast as gRPC + Envoy in terms of goodput and goodput per core. This is because mRPC reduces the number of (un)marshalling steps. The small RPC rate and scalability of mRPC with gRPC marshalling is also shown in Figure 11. Since encoding small RPCs with protobuf is relatively fast, the trend to the rate and scalability is similar to Figure 5a.

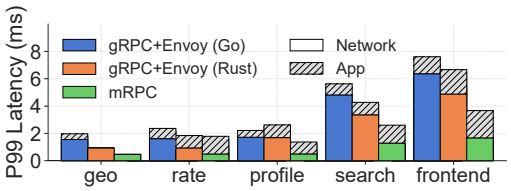


Figure 12: **DeathStarBench**: P99 latency of in-application processing and network processing of microservices, respectively. gRPC with Envoy and mRPC are compared. A null policy is applied for mRPC.

Figure 13: **DeathStarBench**: Mean latency of gRPC without proxy and mRPC.

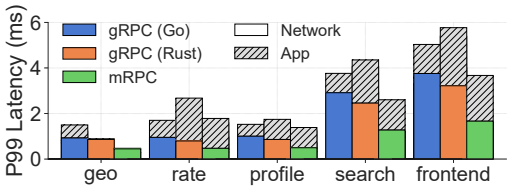


Figure 14: **DeathStarBench**: P99 latency of in-application processing and network processing of microservices, respectively. gRPC without proxy and mRPC are compared.

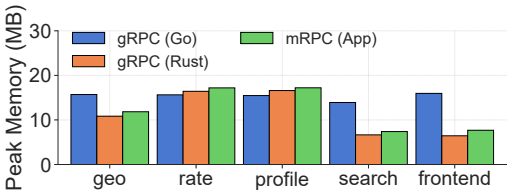


Figure 15: **DeathStarBench**: Peak memory usages of different services. gRPC without proxy and mRPC are compared.

A.2 Extended Evaluation for DeathStarBench

We report the P99 latency of DeathStarBench in Figure 12, comparing gRPC with Envoy and mRPC. The result is similar to the comparison of median latency in §7.4. mRPC speeds up gRPC+Envoy by 2.1× in terms of end-to-end P99 tail latency.

We also evaluate gRPC without proxy and mRPC without any policy enforced. Figure 13 and Figure 14 show the results for mean latency and P99 tail latency. We observe that mRPC speeds up gRPC by 1.7× and 1.6×, in terms of mean latency and P99 tail latency. Communication costs are substantial in the DeathStarBench applications, and thus reducing the communication latency can improve end-to-end application performance. This is consistent with the original

DeathStarBench paper's observation [23].

We further compare the memory usage of gRPC and mRPC. The peak memory consumption of gRPC and mRPC in DeathStarBench applications is illustrated in Figure 15. For mRPC, we report the user application side memory usage, which also includes all the memory pages shared with the mRPC service. We observe that mRPC does not incur notable memory overhead compared to gRPC. On the other hand, we find a small and constant memory footprint of mRPC service across all machines at around 9 MB.