Rethinking RPC Communication for Microservices-based Applications

Xiangfeng Zhu[w]   Yang Zhou[b]   Yuyao Wang[w]   Xiangyu Gao[w]   Arvind Krishnamurthy[w]

Sam Kumar[l]   Ratul Mahajan[w]   Danyang Zhuo[d]

[w]*University of Washington* [l]*UCLA* [d]*Duke University* [b]*UC Berkeley and UC Davis*

Abstract

Fast and efficient RPCs are key to the performance of applications based on microservices. But RPC communication suffers from significant overhead today because it relies on the standard, layered protocol stack and loose coupling between the end host and in-network proxies that process RPCs. We propose delayering the RPC communication stack and tightly coupling the end host and in-network processing using high-level abstractions. This approach leads to more efficient and performant RPC communication because it eliminates many sources of overhead.

CCS Concepts

• **Networks → Programming interfaces**; **Cross-layer protocols**; *Data center networks.*

Keywords
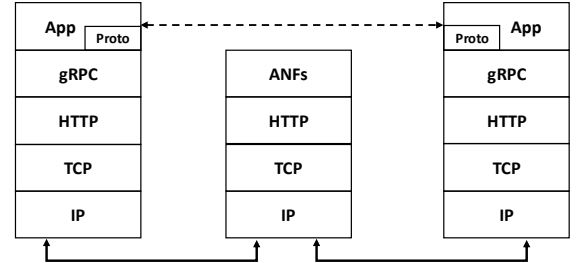
Application Networks, Microservices, RPC

(a) Today's RPC communication.

(b) Proposed RPC communication approach. ANFs can be part of the host processing or in a separate in-network proxy.

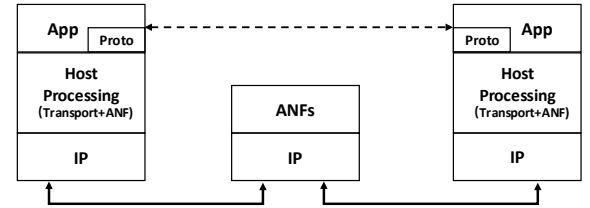**Figure 1: RPC communication approaches.**

1 Introduction

Modern cloud applications consist of hundreds to thousands of microservices, often managed by a single organization or tightly integrated teams. This architecture transforms what were once simple function calls inside monolithic binaries into remote procedure calls (RPCs) over the network.

RPC communication is enabled by exchanging structured data, defined in a language like Protobuf [7], between clients

and servers. It involves serializing the data, adding necessary metadata (e.g., which RPC endpoint was invoked), and delivering it while ensuring end-to-end properties such as reliability and ordering. It also involves applying one or more application network functions (ANFs) for tasks such as request routing, load balancing, security, and monitoring.

Today, RPC communication is typically achieved using a layered stack of standard protocols, as shown in Figure 1a. (We use gRPC [4] as an example; the stack is similar to other RPC libraries such as Apache Dubbo [2].) End hosts rely on gRPC, HTTP, TCP, and IP, and ANFs are implemented as userspace HTTP proxies [3, 6]. Further, developers must manually couple the host stack with ANFs as needed. If an ANF needs to route requests based on the username field of the RPC, the developer must add that information as a custom HTTP header at the end host.

This architecture leads to a significant "RPC tax" [28, 31], stemming from two sources.

Standard, layered end host stack. Today's RPC communications rely on standard and layered end host stacks. This stack bundles features that are unnecessary for certain applications. For example, real-time applications with strict latency requirements may not need in-order or reliable delivery but are still burdened by the overhead of TCP and HTTP/2 [18, 25]. Even within the same application, different RPC endpoints may have different transport requirements—read operations might tolerate relaxed delivery, while write operations demand strict reliability. The interactions between layers can also introduce performance issues. For example, HTTP/2 multiplexes multiple RPCs over a single TCP connection, but because TCP enforces in-order delivery, a single lost packet blocks all subsequent data in the connection, resulting in head-of-line blocking for later RPCs [20]. Lastly, each layer adds its own headers, inflating on-wire messages.

Loosely-coupled end host stack and ANFs. To enable ANFs while remaining compatible with the standard interfaces, service meshes [6, 8, 9] have emerged as a popular solution. In service meshes such as Istio [8] and Linkerd [6], a userspace HTTP proxy is deployed alongside each service instance or as a middlebox, intercepting and processing all incoming and outgoing traffic, and metadata is encoded in HTTP headers [9, 39]. These proxies terminate the TCP connection and parse HTTP headers to implement the ANF's logic. While effective at providing rich functionality, this design introduces substantial overhead—studies have shown that service mesh proxies can increase the latency and CPU overhead by up to 7x [27, 38] due to extra protocol and header parsing, data copies, and context switches. The metadata (HTTP headers) for coupling the end host stack and ANFs further increases header size and parsing overhead.

We propose delayering the RPC communication stack and tightly coupling end host and ANF processing, as shown in Figure 1b. By flattening the stack, we can eliminate redundant features and streamline communication, which can significantly increase performance and efficiency. By tightly coupling end host and ANF processing, we can allow ANFs to directly access RPC data without unnecessary decoding, enabling lightweight, kernel-level, or even hardware-accelerated processing.

To realize this vision, we propose a compiler-based approach that automatically generates optimized RPC communication stacks and on-wire message layout. Application developers specify RPC communication requirements at a high level, including transport properties for each RPC endpoint (e.g., reliability, in-order delivery, priority) and their

ANFs (e.g., policies, routing, observability). Given these specifications and available in-network processing resources (e.g., SmartNICs and programmable switches), the compiler automatically determines which components should execute in the end host stack and which should be delegated to the in-network processor. By understanding how RPC metadata and payloads are used across transport and in-network programs, we can format messages compactly and optimize them for kernel or hardware-accelerated processing.

We can streamline RPC communication in this manner only when both ends and ANFs are controlled by the same organization or closely collaborative organizations, as is common for cloud applications. Our approach can be implemented via a shared RPC library to which different microservices link. The applications may communicate externally as well, with endpoints that use the traditional stack. We support such communication using gateways that are similar to those used by service meshes today [5]. The gateways translate between the standard stack and our custom protocol, and our compiler will generate them automatically.

We are not arguing against software abstractions and modularity, but we are moving abstractions to a higher, specification layer, where we can iterate on them more rapidly. Auto-generating flat implementations from those specifications allows us to avoid the overhead typically associated with direct implementation of layered abstractions.

2 RPC Communication Overheads

Consider a key-value store microservice with two methods: *get* and *set*. The *get* method retrieves the value of a given key, and *set* updates the store with a new key-value pair. For some applications, these two methods may have distinct communication requirements. For instance, consider a load manager that maintains the current load of different Web server replicas. The replicas use *set* to update their load when it changes substantially, while a load balancer periodically uses *get* to retrieve the current load and select a target replica. In this setting, occasional loss or delay of *get* requests or responses may be acceptable, but the loss of *set* requests or responses may result in an extended period of replica overload.

Let us also assume that the application developer seeks to apply two ANFs to all RPCs: (1) an application firewall to block requests from certain users, based on the *username* field, and (2) a session tracker that counts requests per session, based on a *session-id*.

We implement the key-value store application in Go using four different underlying protocols: 1) UDP, 2) TCP, 3) HTTP/2 (which uses TCP), and 4) gRPC (which uses HTTP/2 and TCP). We run the experiments on Ubuntu 20.04 machines with two Intel 10-core Xeon Gold 5215 CPUs and 256 GB RAM. In our experiments, each request or response
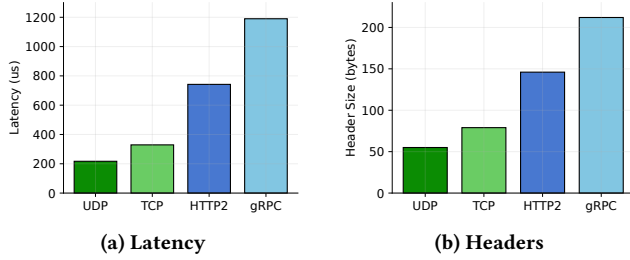
(a) Latency

(b) Headers

Figure 2: Latency and header size for the key-value store using different protocols. All messages in different protocols are serialized with Protobuf.

Figure 3: HTTP header parsing latency.

is approximately 100 bytes, and the workload comprises 80% reads and 20% writes—a typical pattern in caching and metadata-intensive services.

With today's layered and loosely-coupled architecture, the application faces significant performance inefficiencies:

End host stack. Independent of the actual requirements for RPC endpoints, existing RPC communication uses the same heavy-weight stack (i.e., gRPC, HTTP/2 and TCP). Figure 2a shows the performance impact of using different protocols for the key-value store application. Each layer (TCP, HTTP/2, gRPC) adds non-negligible overhead to the end-to-end latency: implementing the *get* endpoint with gRPC increases latency by up to 448% compared to using UDP, a protocol that still satisfies the requirements of *get*.

Further, RPC payloads today are wrapped in multiple layers of headers (IP, TCP, HTTP/2, gRPC), adding substantial header overhead, as shown in Figure 2b. Similar to latency, each layer in the stack adds considerable header overhead. For example, using gRPC adds 133 bytes of header on top of TCP in our experiments. A recent study [35] shows that Twitter's production clusters handle object sizes (key+value) between 55-294 bytes, meaning all headers in a gRPC message can add 42-79% overhead.

ANFs. To implement ANFs like the firewall and session tracker, developers typically rely on service meshes, which intercept traffic through HTTP-based userspace proxies such as Envoy [3]. These proxies implement and exercise the full protocol stack with the outer layers (TCP and HTTP), as shown in Figure 1a: they terminate the incoming TCP connection, parse and extract metadata from HTTP headers, apply the desired ANF logic, and then establish a new connection to the destination service. This design introduces significant overhead due to context switching between kernel and userspace, repeated protocol parsing, redundant data copying, and serialization/deserialization operations. Prior studies [27, 38] show that popular service meshes can reduce
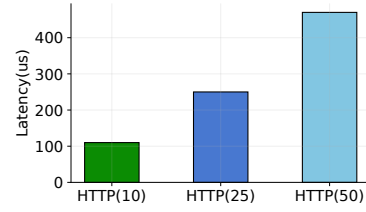
throughput, increase tail latency, and raise CPU utilization by 1.27–7x compared to direct communication.

Worse, because of the loose coupling between service mesh and end host stack, developers must insert metadata required by ANFs as custom HTTP headers at the end host, introducing additional HTTP parsing overhead [38]. This parsing overhead grows with the number of headers, as HTTP parsing scales poorly with increasing metadata. In our study, deployments of Istio [8], a popular service mesh, with observability, security, and routing features typically include over 20 headers per request (including default HTTP/2 headers, gRPC metadata, Istio-internal headers, and tracing fields). As shown in Figure 3, adding 25 custom HTTP headers increases parsing latency by 250 μsecs.

Non-portability. A promising way to reduce overhead is to offload ANFs to the kernel (via eBPF), SmartNICs, or programmable switches [1, 27, 33]. However, offloading is challenging with the current protocol stack because, for any in-network processors, they have to process and manage multiple protocol layers, parse HTTP headers, and buffer the network packets in case the RPC size exceeds MTU (maximum transmission unit). Further, common protocols such as TCP (and its variants) impose reliability and congestion-control mechanisms that complicate the interception and transformation required by ANFs [14].

2.1 Existing Solutions

Prior work has explored these issues in isolation, focusing either on optimizing the end host protocol stack or ANF processing. However, these approaches have had limited success in addressing the root causes of inefficiencies that arise from a layered and loosely coupled architecture.

Optimizing the end host stack alone. Many proposals [10, 17, 18, 20, 25, 32] have focused on reducing the overhead of the transport protocol, particularly TCP. Approaches such as MTP [14] and NetRPC [36] employ custom transport protocols to facilitate in-network computation programs. While these methods effectively reduce transport overhead, they are general-purpose and suffer from excessive features. For example, both MTP and NetRPC guarantee messages are reliably delivered, regardless of application requirements.

NetBlocks [10] allows the application to customize the end host stack and the message layout. However, they fail to address the loose coupling between end host stack and ANFs.

Optimizing ANF processing alone. Other proposals have focused on improving the performance of application networks [1, 12, 22, 34, 39]. For example, Cilium [1] offloads simple, Layer 4 (transport) ANFs to the kernel using eBPF, while Proxyless gRPC [19] and ServiceRouter [30] enable in-process execution of network functions—before the RPC stack at the client and after it at the server. AppNet [37, 39], Lyra [12], and ClickINC [34] introduce new abstractions to compile and optimize function placement. While these solutions achieve performance gains in targeted areas, they are still constrained by the existing layered protocol stack. As a result, significant performance gains from streamlined communication and ANF offloading remain untapped.

Coupling end host stack and ANFs. Recent works on in-network computation co-design the end host stack and ANFs. However, as discussed in §3, they suffer from high development costs and poor reusability.

3 Redesigning RPC Communication

Given the limitations of enabling RPC communication using general-purpose, layered, and loosely coupled architectures, we propose a fundamental redesign that tightly integrates all components of RPC communication. Instead of relying on layered stacks, the end host and ANFs should operate on a flattened, streamlined stack customized for each RPC method. This stack is built directly on top of IP, which provides basic connectivity between endpoints, while higher-level features are tailored to the application's needs.

We advocate for an approach that uses high-level specifications and compilers to enable such RPC communication. To motivate this approach, we outline two alternatives.

Hand-coded, application-specific designs: One approach is to manually design tightly integrated, custom-built systems tailored to specific applications. This approach involves handcrafting each component—serialization, transport, and ANFs—to meet precise performance and operational requirements. Recent in-network computation systems [15, 16, 21, 23, 32] exemplify this category. For example, NetCache [16] introduces a custom message format and leverages a lightweight transport protocol to enable an in-network caching service between the client and the server. This design achieves maximum performance by eliminating unnecessary abstractions and fine-tuning every component to the workload. However, while this approach can deliver great performance, it suffers from high development costs and is difficult to adapt to new use cases or evolving workloads.

Broadened cross-layer interfaces: An alternative approach that prioritizes reusability is to extend the interfaces between protocol layers, enabling richer contextual information sharing. By extending APIs and headers, the serialization layer can communicate performance hints (e.g., priority, consistency requirements) to the transport layer, allowing network functions to adapt dynamically. This approach improves interoperability and preserves the modularity of existing protocol stacks, allowing for incremental deployment without the need for extensive redesign. However, broadened interfaces remain fundamentally constrained by protocol standards, limiting the extent to which they can support end-to-end, full-stack optimization. For example, TCP allows optional headers (referred to as TCP options) to carry additional information, such as window scaling [13] or selective acknowledgments [24]. However, these options are limited by space (40 bytes) and their flexibility.

An approach based on a high-level language and an optimizing compiler offers a balanced solution between the customization of hand-coded systems and the reusability of broadened interfaces. It can deliver the performance benefits of tightly integrated designs without the high development cost and poor reusability of manual solutions. By automatically generating optimized stacks based on application requirements, a compiler-based system can eliminate redundancy and enable seamless co-design of serialization, communication properties, and ANFs. It can also easily adapt to diverse workloads and deployment environments.

4 Key Research Questions

Realizing our approach requires answering a few key research questions.

Q1: *What abstractions should our DSL (domain-specific language) provide to specify RPC communication requirements?* To enable developers to express application-specific communication requirements, our DSL must provide expressive yet concise abstractions. This includes the ability to express end-to-end communication requirements such as reliability, ordering, priority, and ANFs such as routing, load balancing, and access control. The challenge is designing abstractions that are flexible enough to support diverse application needs while still being amenable to efficient compilation and a range of optimizations.

Q2: *How to generate efficient RPC communication stack implementation?* Generating high-performance RPC communication implementations requires translating high-level communication specifications into optimized, low-level code that minimizes the performance overhead. This involves several challenges, such as eliminating redundant processing and parsing and reducing protocol layering. The compiler must also tailor the generated code to the target deployment environment, choosing between user-space, kernel-space, or

in-network execution based on performance and resource constraints.

Q3: *How to enable lightweight ANF execution?* As discussed earlier, ANF execution today has high overhead because the proxies must terminate transport connections and parse HTTP headers, *before* any application-specific functionality is executed. Can we get rid of this overhead while retaining the full expressiveness of ANFs, such as access to message content and the ability to transform or enforce policies on RPCs? Addressing this challenge requires a compiler that can generate custom message formats that expose the necessary metadata to ANFs based on their functionality. We also need to enable in-network processing of in-flight messages while preserving transport semantics and application correctness.

Q4: *How to coordinate end host stack and in-network processors?* Achieving seamless coordination between the end host stack and in-network processors is crucial for efficient execution. This requires mechanisms for partitioning functionality across these components while maintaining correctness and performance. Key challenges include defining clear interfaces for state sharing, ensuring consistency of control and data paths, and dynamically adapting to changes in network conditions or application workloads. The system must intelligently decide which ANFs to offload to in-network devices and which to keep on the end host, balancing resource utilization, latency, and fault tolerance. Effective coordination mechanisms must also handle heterogeneity across programmable hardware and ensure portability across deployment environments. Depending on ANF placement (e.g., on the end host via eBPF or in-network via programmable switches), certain metadata may not need to be encoded in the transmitted message. For example, when implementing a session tracker on the client-side end host, the session ID does not need to be included.

5 Proposed Solution

We outline a potential approach toward realizing our vision, which partially answers the aforementioned questions.

5.1 Programming Abstractions

Developers express RPC communication requirements through a high-level RPC communication specification, which captures both transport properties (e.g., reliability, in-order delivery) and application network functions (e.g., load balancing, access control). We build on the abstractions provided by AppNet [37, 39] and NetBlocks [10] as foundations.

AppNet is a DSL that enables developers to specify ANFs using match-action rules that operate on RPC fields, state variables, and built-in functions across both requests and responses. This model simplifies complex operations such

as mutating, intercepting, reordering, and dynamically modifying RPCs. For the communication specification between each pair of microservices, AppNet comprises a chain specification and corresponding element specifications. The chain specification defines which network functions should be invoked and their order of invocation. Each element specification includes four sections: a *state* section that declares local or shared state variables, an *init* section for state initialization, and *req* and *resp* sections that define match-action rules for processing RPC requests and responses.

A key advantage of AppNet is its deployment transparency: developers are not required to specify where (e.g., end host, SmartNIC, programmable switch) or how ANFs are deployed. The compiler automatically analyzes the input specification and the available infrastructure to determine the optimal placement of functions, ensuring efficient execution across heterogeneous environments.

To further streamline development, we extend AppNet's DSL with NetBlocks for the developers to define end-to-end communication properties alongside network functions using a unified syntax. NetBlocks is a DSL and a compiler for designing ad-hoc protocols. It allows users to configure transport requirements by selecting and customizing features. Similar to NetBlocks, we provide a library of common transport functions (e.g., reliability, RPC ordering) with tunable parameters, allowing developers to customize behavior without implementing these features from scratch.

Figure 4 shows an example specification for the key-value store application.

5.2 Custom RPC Layout and Transport

For lightweight ANF processing, without expensive transport termination or header parsing, our compiler generates a custom RPC layout that enables efficient processing of messages "in flight". Based on metadata usage in the AppNet program and the application-defined RPC structure, the compiler synthesizes a compact, fixed-format message layout with statically known offsets, sizes, and alignment for each metadata field and payload section. This design allows ANFs—whether implemented in eBPF, P4, or userspace—to inspect and manipulate messages cheaply.

In addition, today's transport protocols (e.g., TCP, QUIC) assume immutable, end-to-end byte streams and break down when messages are delayed, mutated, intercepted, or reordered by ANFs. Inspired by MTP [14], our compiler-generated protocol stack includes transport-layer logic that is aware of potential message-level transformations and in-network behavior. For instance, congestion control is ANF-aware and can adapt to delays caused by ANFs. Likewise, reliability is implemented at the message level using end-to-end acknowledgments, avoiding reliance on byte-level sequence numbers that are incompatible with mutable or intercepted messages.

```
1  client:  firewall()->session-tracker()
2  transport:  inorder(strategy="hold-forever")
      ->reliable()
```

(a) A chain specification for the *set* RPC endpoint.

```
1  state:
2    firewall
3
4  init():
5    set(firewall, 'Kevin', 'Yes')
6    set(firewall, 'Peter', 'No')
7
8
9  req(rpc):
10   match get(firewall, get(rpc, 'usrname')):
11     Some(permission) =>
12       match permission:
13         'Yes' =>
14           send(rpc, Down)
15         'No' =>
16           send(err('firewall'), Up)
17     None =>
18       send(rpc, Down)
19
20 resp(rpc):
21   send(rpc, Up)
```

(b) The element specification for firewall.

Figure 4: Example communication specification for the *set* endpoint.

Additionally, the custom transport supports multiple RPCs in the same connection, as in HTTP/gRPC streams. Each packet carries an RPC ID, and the compiler ensures that, when possible, the first packet of an RPC contains all the metadata required by ANFs. This enables early inspection and processing without waiting for the entire message to be received, reducing buffering overhead.

5.3 End host stack and ANF

Given the communication specification and the available processing platforms (e.g., end hosts, SmartNICs, switches), the compiler generates optimized code for both endpoint protocol stacks and ANFs:

End host stack: Based on the communication properties specified for each RPC method, the compiler generates a highly optimized protocol stack at the end host. This stack includes only the necessary features, avoiding unnecessary layers and features, and is customized for each endpoint. One challenge is the inflexibility of the Linux kernel network stack, which limits customizability, whereas userspace stacks

compromise protection [26] and manageability [29]. To address this, we leverage *eTran* [11], a customizable kernel network stack built on eBPF. The generated stack integrates with the XDP layer, a high-performance eBPF hook that processes packets before they reach the socket layer, to ensure low-latency execution.

ANFs: Beyond end-to-end communication properties, the compiler translates AppNet's application network specifications into optimized, target-specific code for in-network processors, for example, P4 for programmable switches and eBPF for kernel-space execution. The generated code is tightly integrated with custom-generated RPC headers, enabling efficient header extraction and processing on in-network processors. When an ANF is placed on a client-side host stack, the metadata used by it will be removed from on-wire messages to reduce header overhead.

6 Conclusion

By tightly integrating RPC communication components via a single abstraction, a compiler can automatically generate an efficient communication stack that eliminates overhead and streamlines data transfer. Additionally, the compiler produces an optimized RPC layout, enabling ANFs to execute efficiently within the network, leveraging emerging kernel and hardware acceleration platforms.

Acknowledgments

References

[1] [n.d.]. Cilium Service Mesh. https://cilium.io/use-cases/service-mesh/. (Accessed on 04/13/2025).

[2] [n.d.]. Dubbo: A Cloud-Native Microservice Framework. https://dubbo.apache.org/. (Accessed on 04/13/2025).

[3] [n.d.]. Envoy. https://www.envoyproxy.io/. (Accessed on 04/13/2025).

[4] [n.d.]. gRPC: A high performance, open source universal RPC framework. https://grpc.io. (Accessed on 04/13/2025).

[5] [n.d.]. Ingress Gateways. https://istio.io/latest/docs/tasks/traffic-management/ingress/ingress-control/. (Accessed on 04/13/2025).

[6] [n.d.]. Linkerd: the world's most advanced service mesh. https://linkerd.io/. (Accessed on 04/13/2025).

[7] [n.d.]. Protocol Buffers. https://protobuf.dev/. (Accessed on 04/13/2025).

[8] [n.d.]. The Istio Service Mesh. https://istio.io/. (Accessed on 04/13/2025).

[9] Sachin Ashok, P Brighten Godfrey, and Radhika Mittal. 2021. Leveraging service meshes as a new network layer. In *Proceedings of the 20th ACM Workshop on Hot Topics in Networks*. 229–236.

[10] Ajay Brahmakshatriya, Chris Rinard, Manya Ghobadi, and Saman Amarasinghe. 2024. NetBlocks: Staging Layouts for High-Performance

Custom Host Network Stacks. *Proceedings of the ACM on Programming Languages* 8, PLDI (2024), 467–491.

[11] Zhongjie Chen, Qingkai Meng, ChonLam Lao, Yifan Liu, Fengyuan Ren, Minlan Yu, and Yang Zhou. 2025. eTran: Extensible Kernel Transport with eBPF. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*.

[12] Jiaqi Gao, Ennan Zhai, Hongqiang Harry Liu, Rui Miao, Yu Zhou, Bingchuan Tian, Chen Sun, Dennis Cai, Ming Zhang, and Minlan Yu. 2020. Lyra: A cross-platform language and compiler for data plane programming on heterogeneous asics. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*. 435–450.

[13] Van Jacobson, Robert Braden, and David Borman. 1992. *TCP extensions for high performance*. Technical Report.

[14] Tao Ji, Rohan Vardekar, Balajee Vamanan, Brent E. Stephens, and Aditya Akella. 2025. MTP: Transport for In-Network Computing. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*.

[15] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. 2018. {NetChain}:{Scale-Free}{Sub-RTT} coordination. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. 35–49.

[16] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. 2017. Netcache: Balancing key-value stores with fast in-network caching. In *Proceedings of the 26th Symposium on Operating Systems Principles*. 121–136.

[17] Anuj Kalia, Michael Kaminsky, and David Andersen. 2019. Datacenter RPCs can be general and fast. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. 1–16.

[18] Marios Kogias, George Prekas, Adrien Ghosn, Jonas Fietz, and Edouard Bugnion. 2019. R2P2: Making RPCs first-class datacenter citizens. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. 863–880.

[19] Steven Landow. 2021. gRPC Proxyless Service Mesh. https://istio.io/latest/blog/2021/proxyless-grpc/.

[20] Adam Langley, Alistair Riddoch, Alyssa Wilk, Antonio Vicente, Charles Krasic, Dan Zhang, Fan Yang, Fedor Kouranov, Ian Swett, Janardhan Iyengar, et al. 2017. The quic transport protocol: Design and internet-scale deployment. In *Proceedings of the conference of the ACM special interest group on data communication*. 183–196.

[21] ChonLam Lao, Yanfang Le, Kshiteej Mahajan, Yixi Chen, Wenfei Wu, Aditya Akella, and Michael Swift. 2021. ATP: In-network aggregation for multi-tenant learning. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. 741–761.

[22] Hao Li, Changhao Wu, Guangda Sun, Peng Zhang, Danfeng Shan, Tian Pan, and Chengchen Hu. 2021. Programming network stack for middleboxes with Rubik. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. 551–570.

[23] Jialin Li, Jacob Nelson, Ellis Michael, Xin Jin, and Dan RK Ports. 2020. Pegasus: Tolerating skewed workloads in distributed storage with {In-Network} coherence directories. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 387–406.

[24] Matt Mathis, Jamshid Mahdavi, Sally Floyd, and Allyn Romanow. 1996. *TCP selective acknowledgment options*. Technical Report.

[25] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John Ousterhout. 2018. Homa: A receiver-driven low-latency transport protocol using network priorities. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*. 221–235.

[26] Simon Peter, Jialin Li, Irene Zhang, Dan RK Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. 2015. Arrakis: The Operating System is the Control Plane. *ACM Transactions on Computer Systems (TOCS)* 33, 4 (2015), 1–30.

[27] Shixiong Qi, Leslie Monis, Ziteng Zeng, Ian-chin Wang, and KK Ramakrishnan. 2022. Spright: Extracting the server from serverless computing! high-performance ebpf-based event-driven, shared-memory processing. In *Proceedings of the ACM SIGCOMM 2022 Conference*. 780–794.

[28] Mubashir Adnan Qureshi, Junhua Yan, Yuchung Cheng, Soheil Hassas Yeganeh, Yousuk Seung, Neal Cardwell, Willem De Bruijn, Van Jacobson, Jasleen Kaur, David Wetherall, et al. 2023. Fathom: Understanding Datacenter Application Network Performance. In *Proceedings of the ACM SIGCOMM 2023 Conference*. 394–405.

[29] Hugo Sadok, Zhipeng Zhao, Valerie Choung, Nirav Atre, Daniel S Berger, James C Hoe, Aurojit Panda, and Justine Sherry. 2021. We need kernel interposition over the network dataplane. In *Proceedings of the Workshop on Hot Topics in Operating Systems*. 152–158.

[30] Harshit Saokar, Soteris Demetriou, Nick Magerko, Max Kontorovich, Josh Kirstein, Margot Leibold, Dimitrios Skarlatos, Hitesh Khandelwal, and Chunqiang Tang. 2023. {ServiceRouter}: Hyperscale and minimal cost service mesh at meta. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. 969–985.

[31] Korakit Seemakhupt, Brent E Stephens, Samira Khan, Sihang Liu, Hassan Wassel, Soheil Hassas Yeganeh, Alex C Snoeren, Arvind Krishnamurthy, David E Culler, and Henry M Levy. 2023. A cloud-scale characterization of remote procedure calls. In *Proceedings of the 29th Symposium on Operating Systems Principles*. 498–514.

[32] Hao Wang, Han Tian, Jingrong Chen, Xinchen Wan, Jiacheng Xia, Gaoxiong Zeng, Wei Bai, Junchen Jiang, Yong Wang, and Kai Chen. 2024. Towards Domain-Specific Network Transport for Distributed DNN Training. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. 1421–1443.

[33] Tao Wang, Jinkun Lin, Gianni Antichi, Aurojit Panda, and Anirudh Sivaraman. 2023. Application-Defined Receive Side Dispatching on the NIC. *arXiv preprint arXiv:2312.04857* (2023).

[34] Wenquan Xu, Zijian Zhang, Yong Feng, Haoyu Song, Zhikang Chen, Wenfei Wu, Guyue Liu, Yinchao Zhang, Shuxin Liu, Zerui Tian, et al. 2023. Clickinc: In-network computing as a service in heterogeneous programmable data-center networks. In *Proceedings of the ACM SIGCOMM 2023 Conference*. 798–815.

[35] Juncheng Yang, Yao Yue, and KV Rashmi. 2021. A large-scale analysis of hundreds of in-memory key-value cache clusters at twitter. *ACM Transactions on Storage (TOS)* 17, 3 (2021), 1–35.

[36] Bohan Zhao, Wenfei Wu, and Wei Xu. 2023. NetRPC: Enabling In-Network computation in remote procedure calls. In *20th USENIX symposium on networked systems design and implementation (NSDI 23)*. 199–217.

[37] Xiangfeng Zhu, Weixin Deng, Banruo Liu, Jingrong Chen, Yongji Wu, Thomas Anderson, Arvind Krishnamurthy, Ratul Mahajan, and Danyang Zhuo. 2023. Application defined networks. In *Proceedings of the 22nd ACM Workshop on Hot Topics in Networks*. 87–94.

[38] Xiangfeng Zhu, Guozhen She, Bowen Xue, Yu Zhang, Yongsu Zhang, Xuan Kelvin Zou, XiongChun Duan, Peng He, Arvind Krishnamurthy, Matthew Lentz, et al. 2023. Dissecting overheads of service mesh sidecars. In *Proceedings of the 2023 ACM Symposium on Cloud Computing*. 142–157.

[39] Xiangfeng Zhu, Yuyao Wang, Banruo Liu, Yongtong Wu, Nikola Bojanic, Jingrong Chen, Gilbert Bernstein, Arvind Krishnamurthy, Sam Kumar, Ratul Mahajan, and Danyang Zhuo. 2025. High-level Programming for Application Networks. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*.