Enabling Configurable, Extensible, and Modular Network Stacks

by

Akshay Krishna Narayan

B.S., University of California, Berkeley (2015)
S.M., Massachusetts Institute of Technology (2019)

Submitted to the Department of Electrical Engineering and Computer Science in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2022

Author: ....
Department of Electrical Engineering and Computer Science
May 13, 2022

Certified by: ....
Hari Balakrishnan
Fujitsu Chair Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by: ....
Leslie A. Kolodziejski
Professor of Electrical Engineering and Computer Science
Chair, Department Committee on Graduate Students

Enabling Configurable, Extensible, and Modular Network Stacks

by

Akshay Krishna Narayan

Submitted to the Department of Electrical Engineering and Computer Science on May 13, 2022 in Partial Fulfillment of the Requirements for the Degree of Doctor of Philosophy in Electrical Engineering and Computer Science

Abstract

Modern networks and the applications that use them are increasingly specialized; each application increasingly uses a bespoke network stack which integrates desired protocols, services, and APIs. This thesis will describe two systems, Bertha and Congestion Control Plane (CCP), which incorporate new abstractions to navigate this new setting from the perspective of congestion control algorithm and the application's network API, respectively. Bertha uses a new abstraction called a Chunnel to represent network services, e.g., hardware offloads of application functionality, publish-subscribe communication services, or encryption. CCP decouples congestion control algorithm implementations from network datapaths by designing an abstract datapath which supports collecting custom measurements and subsequently applying rate or window enforcement.

Thesis Supervisor: Hari Balakrishnan
Title: Fujitsu Chair Professor of Electrical Engineering and Computer Science, MIT

Thesis Committee Member: Mohammad Alizadeh
Title: Associate Professor of Electrical Engineering and Computer Science, MIT

Thesis Committee Member: Aurojit Panda
Title: Assistant Professor of Computer Science, NYU

Thesis Committee Member: Scott Shenker
Title: Professor in the Graduate School, UC Berkeley

Acknowledgments

It is almost unfair that this thesis must list a single name – i.e., mine – as its author. The ideas I describe here are the result of both formal and informal collaboration among friends and colleagues. Any line of work will have its ups and downs; it is one's collaborators, labmates, advisors, mentors, friends, and family who provide the external context, encouragement, advice, technical support, and cheerleading that pursuing research requires. If anyone has come to read this looking for advice, mine is simply this: with a supportive community at your back, you can't really fail.

With this in mind, I first thank my advisors, Hari Balakrishnan and Mohammad Alizadeh, for guiding me throughout my time at MIT. Both Hari and Mohammad have a special talent for finding the core of a research project and highlighting it for everyone to see. One thing I have tried to learn from Hari is prioritization: the art of working on the core of a problem rather than its window dressing. His insistence that research communication should be about teaching the concepts of the work has greatly improved my communication skills. Mohammad, meanwhile, has taught me to explain not just the easy parts of a concept, but the messy edge cases too.

I have had, and continue to have, numerous mentors other than Hari and Mohammad. First among these are Scott Shenker and Sylvia Ratnasamy, who originally encouraged me to pursue research as an undergraduate student. Scott and Sylvia, along with other members of the NetSys lab including Gautam, Justine Sherry, Aurojit Panda, Peter Gao, Rachit Agarwal, Radhika Mittal[1], and Sangjin Han showed me the joy of research and the value of collaboration. Additionally, after I came to MIT, Srinivas Narayana was (and, of course, remains) a valuable source of research and life advice in my first two years. I have been fortunate to work on research projects with multiple of these mentors during both my time at Berkeley and at MIT; Panda and Scott, of course, are even on this thesis's committee. Among these mentors, Panda deserves a special mention; he is equally comfortable discussing low-level DPDK configuration parameters as he is where to take

[1]Radhika later came to MIT as a postdoc, where we worked on Bundler (§4.7.6) together.

the project next, how the operating system's bootloader works, or where the best bakeries and coffeeshops in Boston are. Panda has been incredibly generous with his time during our time working together on Bertha (§3), despite having numerous other projects and students of his own. Finally, last and far from least is Arvind Krishnamurthy, who despite also having students of his own and a full schedule while on industrial leave has made the time over the past year to work with me on Bertha. I have especially appreciated Arvind's ability to identify new ways of applying a project's ideas.

Of course, I have learned not just from my mentors, but from my peers as well. In my first week at MIT I began working with Frank Cangialosi and Prateesh Goyal, and we continued this collaboration across multiple projects: Nimbus [51][2], CCP (§4), and Bundler. I had a wonderful time working with Hongzi Mao and Parimarjan Negi[3] on the Park project. Outside the work this thesis describes, I enjoyed my collaborations with Saksham Agarwal, Lloyd Brown, and Margarida Ferreira, from whom I learned about diverse topics from linear programming to program synthesis.

Outside of my own research, I have benefited enormously from the advice and feedback of the NMS and PDOS research groups in CSAIL at MIT. While Hari, Mohammad, and more recently Manya Ghobadi's NMS research group has been my official home, Frans Kaashoek, Nickolai Zeldovich, and Robert Morris[4] welcomed me into PDOS as well. Some of the many supportive and friendly members of NMS and PDOS (who I have not already mentioned) have vastly enriched my time at MIT include: Sheila, Anirudh, Ravi, Frank W., Malte, Amy[5], Jon[6], Tej, Vikram, Mehrdad, Song-

[2]At the time of this thesis's publication, Nimbus has just been conditionally accepted for publication at SIGCOMM 2022, its 11th submission. I thank and applaud Prateesh, the lead author, for his continued persistence and refusal to give up on this paper.

[3](as well as the quasi-village of authors Hongzi marshalled for the Park project)

[4](and more recently, Adam Belay and Henry Corrigan-Gibbs)

[5]Amy Ousterhout provided feedback and comments on Chapter 2 of this thesis, in addition to numerous other instances of feedback.

[6]My desk was next to Jon Gjengset's in my first year at MIT, and I attribute this as the reason both CCP and Bertha are written in Rust.

tao, Jonathan, Derek[7], Ahmed, Anish, Vibhaa, Venkat, Arjun, Inho, Lily, Josh, Arash, Pouya, Lei, Seo Jin, Ralf, Ariel, Upamanyu, Alex, Pantea, Will, and many others.

Outside of research, my family – Amma, Appa, Rohan, and Madhuri – have always been uniquely able to contextualize my work and remind me of what is truly important, and my friends Sagar Karandikar, Sheevangi Pathak, Shoumik Palkar, and Paroma Varma have entertained me with topics from Formula One to whether checking bags on an airplane is a good idea[8].

My partner Deepti, who I met at MIT while we worked on CCP together[9], has supported me on everything from baking me cookies before a paper deadline to helping me debug my DPDK code. While some have commented that it might be hard to spend so much time with someone who not only shares a career but also is in a closely related research area, I have found that this has only helped us empathize with each other more. I especially admire Deepti's ability to rapidly fill our social schedule with meetups as well as her technical ability to dive deep into a performance optimization.

Finally, thanks to you, for reading this thesis!

[7]Jonathan and Derek warrant additional recognition as my roommates through most of my time at MIT.

[8]It's not.

[9]Another thing I thank Hari for: introducing me and Deepti!

Prior Publication

Parts of this thesis were previously published in part in workshop papers [93, 94] and in conference papers [19, 81, 93].

Contents

1 Introduction **13**
1.1 Bertha: A Runtime Re-configurable Network Stack 16
1.2 CCP: Restructuring Endpoint Congestion Control 17

2 Background **19**
2.1 Evolution of Kernel-Bypass Network Stacks 21

3 Bertha **27**
3.1 Introduction 27
3.2 Related Work 30
3.3 Programming Model 32
3.3.1 The Chunnel Abstraction 32
3.3.2 Bertha Programming Model 36
3.3.3 Optimizations 38
3.4 Runtime Reconfigurability 40
3.4.1 Deciding the Datapath Stack 42
3.5 Implementation 46
3.5.1 Optimizations 47
3.5.2 Bertha Structure 48
3.6 Evaluation 49
3.6.1 Bertha Overheads 51
3.6.2 Microservice Communication 52
3.6.3 Publish-Subscribe Message Queues 54
3.6.4 Sharding and Load Balancing 61
3.7 Conclusion 66

4 CCP 67

4.1 Introduction . . . 67
4.2 Related Work . . . 70
4.3 CCP Design Principles . . . 72
4.3.1 Isolating Algorithms from the Datapath . . . 73
4.3.2 Decoupling Congestion Control from the ACK Clock 74
4.3.3 Supporting per-ACK Logic Within the Datapath . . 76
4.4 Writing Algorithms in CCP . . . 76
4.4.1 Datapath Program Abstractions . . . 78
4.4.2 CCP Algorithm Logic . . . 79
4.4.3 Example: BBR . . . 81
4.4.4 Case Study: Slow Start . . . 83
4.5 CCP Implementation . . . 86
4.5.1 Datapath Requirements . . . 86
4.5.2 Safe Execution of Datapath Programs . . . 87
4.5.3 `libccp`: CCP's Datapath Component . . . 88
4.5.4 Datapath Implementation . . . 89
4.6 Evaluation . . . 90
4.6.1 Fidelity . . . 91
4.6.2 Performance . . . 95
4.6.3 Low-RTT and High Bandwidth Paths . . . 98
4.7 New Capabilities . . . 100
4.7.1 Sophisticated Congestion Control Algorithms . . . 101
4.7.2 Velocity of Development . . . 101
4.7.3 Flow Aggregation . . . 102
4.7.4 Write-Once, Run-Anywhere . . . 104
4.7.5 Park . . . 105
4.7.6 Bundler . . . 107
4.8 Conclusion . . . 121

5 Conclusion 123

1 Introduction

At that time it was also hoped that a clarification of humanity's basic mysteries – the origin of the Library and of time – might be found. It is verisimilar that these grave mysteries could be explained in words: if the language of philosophers is not sufficient, the multiform Library will have produced the unprecedented language required, with its vocabularies and grammars. For four centuries now men have exhausted the hexagons... There are official searchers, inquisitors. I have seen them in the performance of their function: they always arrive extremely tired from their journeys; they speak of a stairway without steps which almost killed them; they talk with the librarian of galleries and stairs; sometimes, they pick up the nearest book and leaf through it, looking for infamous words. Obviously, no one expects to discover anything.

Jorge Luis Borges—The Library of Babel

Computing has evolved over the past several decades from a centralized, monolithic model to the contemporary notion of ubiquitous computing. From the initial network support for the transfer of scientific data between large research institutions to the later rise of cloud computing and mobile computing, the network has become increasingly crucial for the operation of modern computer applications. Indeed, today's networked applications dominate users' experience of using computers, and the near-ubiquitious availability of the network has enabled a large-scale shift towards the use of applications for not only data transfer and messaging, but also conferencing, entertainment, transportation, productivity, finance, and more.

These modern applications must scale to serve not only a large number of users, but also users spanning a vast variety of network envrionments. Applications can communicate with hosts 10μs away (i.e., within a datacenter), or 600ms away (i.e., via a satellite in geostationary orbit). The available bandwidth might be tens or hundreds of gigabits per second in a carrier network or a legacy copper line supporting less than a megabit per second. Thus, building modern networked applications as well as the infrastructure that powers them would not be possible without *layering*, which is closely related to modularity in software engineering. With layering, each component of the broader system is independent and has standardized interfaces to other components; it should thus be possible to swap in alternate implementations of a component without changing other modules or relying on any particular underlying hardware or network environment. Perhaps the most well known example of layering in practice is the use of various physical media in the Internet, which spans WiFi, cellular, copper, and fiber links. Despite the differences in how these technologies achieve their shared core goal of packet delivery, the layered abstractions the Internet has adopted enable applications to use the network without special consideration for each underlying transmission media.

The way this type of layering manifests in networked applications is in the *network stack*, the software component that exposes the network's hardware resources to the application. Traditionally, this network stack has been a component of the operating system. While this traditional networking stack has remained stable for three decades—thanks in part to its use of layering—a recent trend toward more in-network services and supporting set of new network hardware has transformed networking stacks. These network services provide a rich set of features such as load balancers to support scalability or hardware acceleration of encryption. Unfortunately, the trend toward more featureful networks has not preserved layering, resulting in increased complexity and unstructured extensions. For example, individual libraries are tied to certain environments (such as a specific cloud provider's network), and incorporating new hardware can involve reimplementing large parts of the code. Especially today, with applications relying on third-party libraries and network features, changing an application's extended network stack–including not only the operating system's traditional packet transport functionality but also modern libraries which facilitate the use of network services and hardware–is considered daunting enough to cause "lock-in" to cloud networks and libraries. Indeed, losing modularity means that work on powerful network services (e.g., in-network hardware offloads of application features) risks not seeing deployment if using those features would complicate an application's development or deployment.

This thesis thus explores the following question:

> How should the adoption of modern networking features affect the way we build networked applications as well as network datapaths?

Ideally, we would re-design the networking stack to regain modularity and extensibility, so that developers can build correct, efficient, and portable applications. This thesis thus addresses this need in two ways:

- Bertha (§3) is a new *runtime re-configurable* way of building networking stacks that allows applications to decide at runtime which implementations of their desired network features to use.

- Congestion Control Plane (CCP) (§4) is a new way to implement congestion control algorithms, a key datapath component, in a modular and portable way across different network stacks.

1.1 Bertha: A Runtime Re-configurable Network Stack

Applications today must use "communication libraries" to bridge the gap between their code and network services, but this approach has led to both environment lock-in and low expressivity. Because application developers must commit to a single library's implementation of each feature, they cannot easily move their application to a different environment (say, a different cloud provider's network) if the library is tied to a runtime environment. Further, with libraries that support only a specific set of other libraries (e.g., an RPC library supporting only certain TLS "backends"), developers are limited in their choice of implementation for the feature their application needs. Imagine instead that applications could provide specifications of network functionality they wanted without committing to a specific implementation, and then selected at runtime the best implementation that fit the available network envrionment. Just as compilers today apply environment-specific optimizations when deciding which instructions to include, the network stack could select network feature implementations best for the network environment. In this case, applications would be both portable and efficient: their high level feature specifications could apply to any network, but they would still use optimized feature implementations and thus get good performance.

Bertha, a runtime re-configurable network stack, provides this capability. Bertha's core abstraction, called the Chunnel, specifies the use of a network feature such as load balancing without committing to a specific implementation. Application developers can easily compose Chunnels into *Chunnel stacks* that specify the network functionality they need. Bertha then decides when establishing connections at runtime which implementations of each network feature to use. This “runtime re-configuration” enables optimizations that are not possible otherwise; e.g., a multicast application can switch Chunnel implementations depending on the number of connection participants to provide better performance.

Further, because Bertha makes the Chunnel stack explicit and type-checkable, domain experts can publish common optimizing transformations over Chunnel stacks which application developers can then import and use. For example, a domain expert might have the insight that a microservice application that uses a container networking interface (CNI) that encrypts inter-container network traffic by default does not require the additional use of an application-level encryption library. This domain expert could then publish a rule-based transformation that developers could apply to their Chunnel stacks to remove the redundant library.

This thesis demonstrates three applications built with Bertha which demonstrate its usefulness in making network features and services more accessible: a microservice application with a virtual network bypass for machine-local connections, a publish-subscribe queue that dynamically transitions implementations depending on the number of participants, and a sharded key-value store that moves the sharding functionality between the client and server at runtime.

1.2 CCP: Restructuring Endpoint Congestion Control

Application developers must target a wide range of network environments. As networks have grown more dynamic, developers have changed their network stacks in two ways. First, they have adopted special-purpose network stacks based on QUIC, kernel-bypass, or hardware processing. Additionally, developers increasingly employ new congestion control algorithms to more efficiently and fairly utilize available network bandwidth. Many such proposals use sophisticated techniques including machine learning or signal processing that are difficult, if not impossible, to implement in a tightly-constrained datapath environment where performance is crucial. These two trends combine to create a problem for both datapath developers and congestion control researchers: on one hand, incorporating new congestion control algorithms into a datapath requires knowledge and effort per each congestion control algorithm, and on the other hand congestion control researchers must expend per-datapath effort to implement their algorithms so applications can use them.

To address this challenge, Congestion Control Plane (CCP), a system in which the datapath can measure common congestion signals for ongoing connections and periodically provide this to an off-datapath module that encapsulates the congestion control algorithm's implementation. With CCP, both datapath support for arbitrary congestion control algorithms and implementing new congestion control algorithms become one-time efforts. A core component of CCP is its datapath language with which congestion control algorithms can safely specify per-packet logic (e.g., calculating measurements or triggering state transitions) atop the datapath's base measurements.

2 Background

The adoption of modern networking features, including hardware and new applications, has caused a divergence in *network datapaths* that applications use. The network datapath is the set of libraries and hardware the application uses to transfer data from its internal logic to and from the network. This chapter discusses the divergence of modern datapaths over time away from shared functionality provided by the operating system towards application-specific libraries that bypass the operating system altogether.

Traditionally, access to the network has been mediated by the operating system. Just as the operating system mediated access to other hardware resources such as memory, storage, and input before the adoption of computer networks, once computers gained networking features the natural place to provide access to them was in the operating system. Over time, the operating system's network stack evolved to gain more features and support a wider range of applications. While the abstractions the operating system has provided have been useful, they have come at high cost: the operating system's interrupt-driven structure optimizes for better multiplexing and efficiency, and sacrifices latency to do so. However, a positive aspect of the operating system's network stack has been its standardized API and structure. For example, when network line rates first began increasing dramatically, the Linux kernel was able to make widely-adopted changes to support a wide range of applications, such as the shift to an initial congestion window of 10 packets [34]. Once this change was made, a wide variety of applications could benefit from it, since almost all applications used Linux as their datapath.

In the past decade, network line rates have continued to scale while the amount of available compute has stagnated. Further, the servers on which applications run are more often highly loaded, so the traditional interrupt-driven approach has not saved much energy. Finally, as applications scaled to run across many machines, the end-to-end response time to a request could depend on many component requests internally, and the slowest of these requests would dominate the end-to-end response time. This problem was described as the "Attack of the Killer Microseconds" [9]. As a result, there has been a trend towards a new way of building and running network applications: running servers "hot" (i.e., at high CPU load), bypassing the operating system's network stack in favor of accessing network hardware directly from user-space, and using spin-polling instead of interrupts to access hardware. This trend began with proposals such as Netmap [114] and PacketShader [56], and gained broader adoption with Intel's DPDK [32]. Since then, a number of research systems have leveraged DPDK, or other kernel-bypass networking technologies such as RDMA, to provide low-latency network stacks to applications. A few initial efforts specialized to key-value store applications were MICA [78], FaRM [33], Pilaf [91] and HERD [68]. Beyond these, eRPC [69] showed that these optimizations can generalize to other RPC-structured applications, and mTCP [63] proposed a scalable TCP implementation for such applications to use. While high-performance computing environments have long proposed even more specialized systems and programming environments which rely even more on hardware support [27, 59, 119], this thesis instead focuses on how applications running on general purpose computing hardware can access the network. Therefore, while recent efforts such as:

- Tonic [6] proposes offloading TCP functions to the NIC.
- TAS [71] streamlines applications' TCP processing on a dedicated set of cores.
- Floem [108] provides an API to more easily offload application components to SmartNICs.
- NanoPU [61] moves the entire datapath into hardware.

have demonstrated performance benefits, this thesis focuses instead on software datapaths.

2.1 Evolution of Kernel-Bypass Network Stacks

While these new high-performance datapaths were at first tightly integrated with the application, over time, they gained features to support multiple applications together, since it is important for deployment that applications are able to multiplex onto machines. To achieve this goal, it is important to understand *why* traditional datapaths could not support the performance applications have demanded. At a high level, sharing data across cores remains expensive, and it is important for performance to avoid it. At the same time, scaling datapaths and applications horizontally to use many cores is important to make use of available resources, so it is often necessary to support cross-core data sharing. For example, Cai et al. [16] explore potential changes to the Linux kernel network stack to scale it to 100 Gbit/s access link bandwidth. They observe that a single core cannot saturate 100 Gbit/s links and suggest scaling certain elements of the stack, including data copies, to multiple cores. There are three challenges the kernel's network stack has faced in offering high throughput and low latency for short message-oriented connections to applications:

- **Flow steering.** While packets arrive from the network in a single place (the NIC), to be accessible to application logic they must[1] then move to the CPU. The datapath must therefore make a decision: which core should it send the packet to? As MegaPipe [57] observed, this is a significant source of inefficiency in the Linux networking stack. Ideally, packets corresponding to the same flow would go to the same core, since it is often possible to isolate memory per-flow to avoid sharing. In this case, flows can be partitioned across cores, and the only remaining cross-core accesses would be on application state. However, Linux's networking stack often cannot perfectly partition flows across cores, especically struggling with cross-core contention on the `accept()` queue of new connections, and requiring cross-core accesses when send-side and receive-side operations are split across cores.

- **User-space to kernel-space transitions.** While useful, system calls ("syscalls") and the need to transition between kernel and user address spaces are expensive, and any networking stack the operating system provides must use syscalls. More recently, efforts in Linux such as `io_uring` have attempted to address this by introducing batching and other optimizations. However, removing syscall overheads has been a key concern for many new datapaths.

[1]As above, while some proposals question this necessity by moving parts of the application to the NIC, here we consider software datapaths only.

- **Extensibility.** While having a common operating system datapath was useful for extensibility (e.g., the change in initial window discussed above), it was hard to make the datapath more featureful. Modern application datapaths such as QUIC [76] rely less on features the operating system datapath offers, choosing instead to use userspace implementations of modern networking features like encryption.

Over the past decade, various new proposed datapaths have attempted to address each of these three concerns. **Arrakis** explores the implications to the operating system once the primary datapath for applications moves into user-space. It argues that the operating system is still valuable as a control plane to mediate access to network resources. Rather than using DPDK, it implements a custom network datapath called "Extaris" which communicates with the device driver directly. It uses interrupts triggered by hardware doorbells to deliver data to applications, and integrates this with the scheduler to wake the application if it is not running. Arrakis's primary concern is to absolutely minimize the amount of shared state required in the datapath, and it takes this design decision to an extreme: the datapath statically partitions the network address space among its applications. This way, it can take advantage of SR-IOV, which supports mapping NIC queues to different data structures. With SR-IOV and address space partitioning, Arrakis can ensure that no datapath state is shared across cores.

More recently, other systems (summarized in Table 2.1) have focused on minimizing user-space to kernel-space context switches while still allowing some cross-core data sharing. IX [10], ZygOS [110], Shenango [102], and Demikernel [141] fall into this category. Where these systems differ is their approach to horizontal scaling:

- **IX** focuses on the idea of run-to-completion on batches of packets.

Datapath	Approach
IX	Batching, run-to-completion
ZygOS	Work stealing via IPI, intermediate queues
Shenango	Integrated scheduling for core allocation
Demikernel	API design for multiple hardware backends

Table 2.1: Summary of the characteristics of several recently-proposed high-performance network datapaths.

- **ZygOS** uses work stealing via inter-processor interrupts (“IPI”) to distribute requests across cores. ZygOS uses insights from queueing theory to distribute requests among cores, no matter which core they originally arrived on. This limits the tail latency that applications using ZygOS observe.

- **Shenango** introduced the idea of “CPU efficiency” and datapath-integrated core allocation. Shenango uses an “iokernel” to distribute packets to application threads, and manages applications’ core allocation to ensure that applications which need to process packets have enough cores allocated to do so. If an application is falling behind on its packet queue, the Shenango/Caladan scheduler will allocate more cores for that application to prevent the queues from growing. Unlike ZygOS, which dedicates a fixed number of cores to each application, Shenango and Caladan vary each application’s core allocation dynamically. Thus, Shenango and Caladan are able to (unlike ZygOS) allocate compute to bulk processing applications when latency-sensitive applications offer little load, but then quickly reclaim those resources to maintain low latencies when offered load increases. Caladan [44] extends this idea of scheduling to other hardware resources (not only the network).

- **Demikernel** [141] offers a “library operating system” approach so that applications can be configured at compile-time to work across multiple different kernel-bypass network technologies, such as DPDK or RDMA.

The final category of systems focuses on improving extensibility. For example, **Snap** [83] and its component network stack “Pony Express” focus on enabling rapid development and deployment of networking features on high-performance network stacks. Snap scales out individual packet processing “engines” which encapsulate dataplane operations.

The systems this thesis describes target extensibility and ease of programming in a different sense. Modern applications are now faced with a wide range of datapaths to choose from, each of which provides different performance characteristics for different application workloads. As a result, it is unlikely that we will see a re-convergence of applications toward a standardized API and structure. CCP brings extensibility to a type of datapath functionality that is useful to have *across* datapaths, while Bertha can bridge different datapaths' APIs and programming models to support portability, as well as the new capability of runtime reconfigurability.

3 Bertha

3.1 Introduction

Today's networked applications rely on a variety of communication libraries, including fast network I/O libraries (e.g., DPDK, IB verbs, and Shenango), RPC libraries (e.g., gRPC, Thrift, and Cap'n Proto), TLS and other encrypted communication libraries (e.g., OpenSSL, BoringSSL, and nqsb-TLS [70]), libraries implementing publish-subscribe and other group communication patterns (e.g., ZeroMQ and Gloo), and libraries that use cloud-provider communication services (e.g., Boto or AzureSDK). Communication libraries have played a key role in allowing users to benefit from new and emerging network functionality (enabled by trends including growing network capacity, deployment of programmable network hardware, and complex network software) and in ensuring that new network services can be deployed and used without requiring changes to the OS kernel or the network stack. However, there is no agreed-upon convention or specification for the interface (API) a communication library should provide, and in practice, APIs differ widely between libraries. Even libraries that implement seemingly similar functionality (e.g., OpenSSL and nqsb-TLS or gRPC and Thrift) often have very different APIs. These interface differences pose an impediment to *application portability*, limit the amount of *runtime recon-*

figurability an application can offer, and limit *composability*.

Application portability.. Application developers use accelerated network I/O libraries, e.g., DPDK, Shenango or IBVerbs, to decrease communication latency and increase throughput. However, these libraries impose constraints on what hardware and network an application can be deployed on (e.g., requiring a DPDK-compatible NIC or RDMA network). As a result, applications that use these libraries only work on specific servers and instance types, and must be rewritten (to use a different network I/O library) to be deployed in other locations. For example, when deploying a Shenango application on AWS EC2, the user must choose an "Enhanced Network Adapter"-enabled instance, and the application cannot be deployed on many popular instance types, including all "T2" instances. As a result, the lack of common interfaces between these libraries impedes *application portability*, requiring the application to be rewritten rather than merely recompiled or even simply re-run when deployment scenarios change.

Runtime Reconfigurability.. When deciding how and where communication functionality should be implemented, application developers often need to trade off functionality, performance, scalability, and efficiency. For instance, consider a client-server application that requires sharding to serve all clients. Alice, the application developer, can choose to implement sharding on the client (i.e., the client directly sends requests to the correct shard) or in a middlebox. These choices provide different benefits and impose different limits: while the client implementation is the most scalable option, it increases the client's processing burden with potentially impacts battery life. Similarly, while the middlebox implementation is less scalable and potentially adds to Alice's cost, it minimizes client requirements. The relative utility of these choices varies with the client and scenario: while the middlebox-based approach is ideal when the client is energy-constrained, (i.e., running on a cellphone or on a battery-powered laptop), the client approach is ideal for energy-unconstrained devices (e.g., desktops or wall-powered laptops). Application developers today must either (i) choose an option when writing their application, or (b) maintain two mostly independent code paths to switch dynamically. In the latter case, the non-uniform interfaces communication libraries provide limit the ability to share code between the configurations and enable runtime reconfigurability.

Composable functionality.. Finally, many communication libraries implement their functionality using other libraries. For example, RPC libraries, including gRPC and Thrift, use BoringSSL or OpenSSL to open TLS connections. However, because current libraries offer non-uniform interfaces, the *application developer* (as opposed to the library developer) has little or no control over the full set of communication libraries their application uses. For example, consider Alice, an application developer who is worried about bugs in TLS libraries and thus wants to make sure that all TLS connections established by her application use `nqsb-TLS` [70], a verified TLS library. While Alice can easily enforce this constraint for any connections her application opens directly, she cannot easily reconfigure gRPC to use `nqsb-TLS`. Furthermore, because OpenSSL and `nqsb-TLS` implement different interfaces, changing gRPC to use this library would require that Alice rewrite it, a significant endeavour for an application developer. A common interface would significantly simplify this problem, allowing Alice to easily swap out OpenSSL and replace it with `nqsb-TLS`.

In this chapter, we propose Chunnels, a new abstraction that provides a common interface for accessing communication functionality. We show, using a variety of examples, that this abstraction addresses the three challenges we highlighted above without limiting the types of communication functionality they can offer. Our evaluation (§3.6) shows that the Chunnel abstraction does not significantly harm application performance. Furthermore, we show that the Chunnel abstraction enables reasoning about the communication functionality a connection uses and that this information can be used to automatically optimize application performance.

We have implemented the Chunnel abstraction in a framework called Bertha. In addition to implementing the Chunnel abstraction, and several Chunnels, Bertha also implements a *negotiation protocol* that enables runtime reconfigurability. Bertha's negotiation protocol not only enables runtime reconfigurability but also allows Bertha applications to detect and avoid bugs resulting from incompatibility between two communicating endpoints. We have implemented several applications using Bertha, and in this chapter, we use these applications to evaluate the performance overheads of Chunnels and to demonstrate the benefits of the Chunnel abstraction.

We begin by reviewing related work before providing details about Chunnels and Bertha's programming model.

3.2 Related Work

Bertha is related to and inspired by prior work in implementing extensibility and optimization.

Extensibility.. The motivation for Chunnels is similar to those for techniques in operating systems to support the development of new types of hardware. The Linux kernel's device driver model [92], syscalls [47], and networking stack [129,130] have all been targets for proposals targeting extensibility. Similarly, Click [73] and OVS [131] describe extensible designs for software routers. Unlike these systems, Bertha picks between feature implementations at runtime. The x-Kernel Protocol Framework [60] previously proposed an architecture for an extensible network stack. Similar to Bertha, the x-Kernel stack supported extensions, but these modules implemented protocols (e.g., variants of TCP) rather than Chunnels' communication functions. While both systems are extensible, the type of extensibility differs. Finally, the Microsoft IIS web server library [90] expresses applications as a combination of high-level application logic and semantic modules. In databases, Volcano [52] and EXODUS [21] proposed an architecture for an extensible query execution system to support new functionality such as query optimizers.

Several efforts have also discussed network APIs and extensions that can better support complex network devices, including from active networking [125] to Netcalls [120], DOA [133], and Serval [98]. While these approaches allow users to explicitly run code in network devices, Bertha exposes slices of network functionality that applications are already using in a structured way. In this vein, Freimuth et al. [43], Eran et al. [37], and Pismenny et al. [109] proposed modifications to network stacks which enable them to take advantage of offloads. While these approaches show promise and are compatible with Bertha, Bertha's goal is different: to allow safe composition of mutually-unaware communication libraries.

Optimization.. The idea of implementing optimization passes as structured translations is widespread. LLVM optimization passes [79] are implemented as IR-to-IR translations. Kohler et al. [72] adopted a similar approach to optimizing Click router configurations. OPT++ [67] and EROC [86] express database optimizations as SQL rewrites. In machine learning, Tensorflow [49], ONNX [100], TVM [23], and FlexFlow [64] rep-

Chunnel	A specific piece of network functionality.
Chunnel stack	An application's specification of the set of Chunnels it wants to use.
Datapath stack	The set of Chunnels Bertha chooses among those in the Chunnel stack.
Optimization Pass	Rule-based compile-time modification to the Chunnel stack.

Table 3.1: Glossary of terms used in this chapter.

resent models as semantic DAGs and apply optimizations before mapping them to available hardware. Finally, in data analytics, Weld [104] enables cross-library optimizations by representing programs in a common IR.

3.3 Programming Model

We next detail our programming model. First, we describe the Chunnel abstraction and the programming interface that library developers (represented by "Chani Chunnelbuild" below) use to implement communication functionality. We then describe the Bertha programming interface that application developers ("Alice Appwright") use to create connections that use these Chunnels. Finally, we describe Bertha's optimization interface that developers ("Ophelia Optimizestack") use to develop *reusable optimization passes* that allow developers such as Alice to quickly and safely optimize their applications.

3.3.1 The Chunnel Abstraction

The Chunnel is the core abstraction we propose: it represents a single communication function, i.e., logic that can transform data (e.g., serializing, en-

```rust
// The Chunnel control path.
pub struct AChunnel { /*...*/ }
// The ChunnelTransformer<R> trait implements connection
↪  establishment logic.
impl<R> ChunnelTransformer<R> for AChunnel
where R: ChunnelDatapath</*...*/>> {
  // Specify that AChunnelDP is the datapath used
  type Connection = AChunnelDP<R>;
  // Chunnel composition interface: compose AChunnel with inner.
  fn connect_wrap(&mut self, inner: R) ->
    Self::Connection {/*...*/}
  // Specify relative compat. for runtime reconfig.
  type Capability = /*..*/;
  fn capabilities() -> Self::Capability { /*...*/ }
}
impl AChunnel {
  // Create a new AChunnel
  pub fn new(/*...*/) -> AChunnel { /* ... */}
}
// The AChunnel datapath
pub struct AChunnelDP<R>{/*...*/}
impl<R> ChunnelDatapath for AChunnelDP<R>
where R: /* input data type requirements */ {
  type Data = /*..*/;
  fn send(&self, msg: Self::Data) { /*..*/ }
  fn recv(&self) -> Self::Data { /*..*/ }
}
```

Listing 3.1: The Chunnel interface.

crypting, or compressing data), can decide where to send data (e.g., which shard or DHT node should receive a request), or replicate data to multiple endpoints (e.g., to implement publish-subscribe functionality). Application developers, e.g., Alice, do not directly call into Chunnel code. Instead, as we explain in §3.3.2, they create connections using a *stack* of Chunnels, and send and receive data over these connections. This stack of Chunnels determines how the application's data is routed and processed. While we do not limit the functionality a Chunnel can implement, we assume that Chunnels are atomic specifiers of functionality. This is in contrast to how applications use communication libraries such as OpenSSL, where the library offers hundreds of available features; in our model, such a library would offer several distinct Chunnels.

We had three goals when designing the Chunnel abstraction: (a) A unified interface: as we argued in §3.1, the lack of unified interfaces is an impediment to portability, runtime reconfigurability, and composition; (b) Generality, i.e., capturing most communication functions, including ones that modify, route or replicate data; and (c) *Safe composition*, i.e., preventing application developers from composing incompatible Chunnels. As we describe next, we meet these goals by carefully designing Bertha's Chunnel programming interface. We implemented Bertha in Rust (§3.5), and describe our interfaces using Rust code. However, the core ideas and abstractions we describe here are general and can be implemented in other languages.

Bertha's Chunnel programming interface (Listing 3.1) requires Chunnel developers, e.g., Chani, to implement a control interface (`ChunnelTransformer`, line 4) and a data interface (`ChunnelDatapath`, line 20). Bertha uses the `connect_wrap` function (line 9) to compose Chunnels in a connection. If Chunnel A appears above (i.e., is pushed after) Chunnel B in a connection's stack, Bertha calls Chunnel A's `connect_wrap` function with an `inner` argument that is a connection including Chunnel B's functionality. Chunnel A's `connect_wrap` function will then return a new connection whose data is first processed with Chunnel A's functionality and then passed to Chunnel B. By recursively calling `connect_wrap` on the Chunnels in the connection's Chunnel stack, Bertha creates a connection that incorporates all of the application's required functionality. As we describe below in §3.3.2, the first `inner` argument Bertha uses for the Chunnel at the bottom of the stack is a *base connection* that provides basic connectivity and is implemented using an underlying network datapath, e.g., the socket library or Shenango [102]. The control interface's `capabilities` aids runtime reconfigurability, and we thus discuss it in §3.4.

The data interface provides functions that Bertha applications use to send or receive data over a connection that uses the Chunnel. Along with specifying how data is processed and routed, this interface is also responsible (line 22 and 23) for specifying the Chunnel's input and output data types. We use this type information to check composition safety at compile time (by specifying constraints on `R` on line 22). For example, a Chunnel that implements sharding needs access to a key that it can use to identify the correct shard. Such a tunnel can specify an input type such as `(String, Vec<u8>)` to indicate that it requires a string key to be passed along with the data. Bertha and the Rust compiler use this information to ensure that if an application developer composes another Chunnel (e.g., a serialization Chunnel) with the sharding Chunnel, then the first Chunnel preserves the key.

Thus, the Chunnel control and data interfaces enable safe composability. Further, as we demonstrate later in §3.6, we have successfully implemented a variety of different communication functions using this interface, thus demonstrating its generality. We next discuss the Bertha application programming model, which provides application programmers with a common interface for using Chunnels.

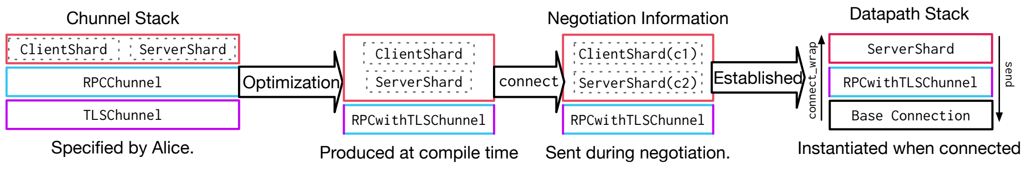


Figure 3.1: An overview of Bertha's compile and runtime processing steps.

```
1  // Alice provides application code.
2  let shrd = ClientShard::new(cfg);
3  let ser = SerializeChunnel::new(idl);
4  let conn = tbm::make_stack!(shrd, ser)
5        .connect(kernel::socket(), addr);
```

Listing 3.2: The code used by an application developer to create and use a connection.

3.3.2 Bertha Programming Model

As we described above, application developers like Alice specify a Chunnel stack when creating a connection, and the behavior of the connection is determined by that stack's composition and ordering. We show the code that Alice writes to construct a connection in Listing 3.2. On lines 2—4 she creates a connection that composes a client sharding and serialization Chunnel. Observe that Chunnel developers can require users to specify additional configuration information when creating a Chunnel; in this case, Alice specifies the sharding configuration (`cfg`) and how data should be serialized and deserialized (`idl`). Next, on line 3, Alice composes these two Chunnels into a connection with the `make_stack!` macro. Here, the call specifies that any data sent using this connection should be first processed by the sharding Chunnel (`shrd`) – which can determine where data should be routed – and then serialized Chunnel (`ser`). Next, on line 4, Alice connects to the server (at `addr`). As a part of this call, Alice specifies that she wants to use the kernel network stack (`kernel::socket`) as the base connection; to use a different networking stack, Alice would simply pass a different base connection. Beyond the code that we show, Bertha's application interface also allows Alice to specify a set of optimization passes (§3.3.3) that she wants to apply to her connections.

Given this code, Bertha performs the following compile- and run- time operations to establish a connection (Figure 4.1): First, at compile time, Bertha tries to apply each of Alice's specified optimizations to the connection's Chunnel stack. As we explain next, optimizations can output new Chunnel stacks that should be used instead of the stack the application developer specifies. Subsequently, the compiler generates a binary where all connections use these optimized Chunnel stacks. At runtime, when the program opens a connection (by calling `connect`), Bertha first uses the specified network I/O library (the socket API in Listing 3.2) to establish a base connection. Next, the Bertha runtime executes its negotiation protocol (§3.4) to perform any runtime configuration and to check that both endpoints are using a compatible set of Chunnels. Next, Bertha makes recursive calls to `ser` and `shrd`'s `connect_wrap` functions, and passes in the base connection as input. The application sends and receives data using the connection returned by this recursive call.

Summary.. The Bertha API and runtime allow us to achieve our composability, portability, and runtime reconfigurability requirements. Our API achieves composability by allowing programmers to specify a Chunnel stack and thus composing functionality from different Chunnels. Chunnel developers can use a Chunnel's input and output data types to constrain some compositions; these constraints are necessary to ensure correctness. To

```
let conn1 = tbm::make_stack!(
     RpcChunnel::new(/*...*/),
     TLSChunnel::new(/*...*/));
let conn2 = tbm::make_stack!(
     RpcWithTLSChunnel::new(/*...*/));
// Replace occurrences of RpcChunnel with TLSChunnel,
// with a better performing RpcWithTLSChunnel
#[tbm::Subst(if [*, RpcChunnel, TLSChunnel, *]
             then [*, RpcWithTLSChunel, *])]
pub trait FuseSerializeAndTLS {
  type Opt;
  fn fuse_tls_serialize_opt(self) -> Self::Opt;
}
```

Listing 3.3: A Bertha optimization pass.

achieve portability, our API requires programmers to specify the network I/O library to use to create a base connection, and switching I/O libraries requires only changing this parameter. In practice, we expect that the code itself will use conditional compilation to allow users to easily decide which network I/O library should be used when building an application, and thus porting applications across deployments would merely require rebuilding them. Finally, Bertha's negotiation protocol, which we explain in §3.4, provides runtime reconfigurability.

3.3.3 Optimizations

Similar to applications built using communication libraries, the performance and resource requirements of a Bertha connection depends on its Chunnel stack. In some cases, two Chunnel stacks that provide identical features might provide very different performance. For example, consider a case where Chani, a Chunnel developer, creates a set of Chunnels that expose functionality from the gRPC library. In this case, Chani might implement two Chunnels: an `RpcChunnel` that uses unencrypted connections and a `RpcWithTLSChunnel` that leverages gRPC's built-in TLS support to provide encrypted connections. Given these Chunnels, imagine our application developer Alice wants a connection that sends encrypted RPC traffic. Alice could either use the method `conn1` of Listing 3.3 (Line 1) shows (composing the unencrypted `RpcChunnel` with a `TLSChunnel`), or she could use the method `conn2` (Line 4) shows, using a single `RpcWithTLSChunnel` Chunnel. While both provide equivalent functionality, `conn2` has better performance because it allows Chani and the gRPC developers to perform cross-layer optimizations. As a result, application developers like Alice should usually (though, as we explain below, not always) prefer the approach used by `conn2`.

When Alice learns about this optimization, perhaps by reading a blog post from Ophelia (a Chunnel developer or domain expert), she might want to apply it to her application. A naive approach for doing so would be for her to audit her application code, find all instances of Chunnel stacks composing an RPC Chunnel and a TLS Chunnel, and replace them with the combined Chunnel. This is a tedious process, and Alice must correctly make this change for each connection that uses this Chunnel stack. We observe that Bertha's interface, which makes a connection's entire Chunnel stack explicit and well-structured, provides us with an opportunity to automate this process.

We show Bertha's approach to automating optimizations on Listing 3.3, Line 8. In this case, along with her blog post, Ophelia can publish an *optimization pass*. The optimization pass specifies (via the `Subst` macro, described in §3.5) that a sequence of Chunnels (in this case RPC followed by TLS) should be replaced by another sequence (in this case RPC-with-TLS). In our implementation, optimization passes are Rust procedural macros and are applied at compile time. These optimizations analyze all Chunnel stacks in the application, and make substitutions when appropriate. In the example above, this allows Alice to adopt Ophelia's optimization by linking against Ophelia's optimization pass.

Bertha assumes that optimization are correct; as long as this is the case, the interface we have described preserves application safety properties. For example, Alice might require all encrypted connections to use nqsb-TLS. In our programming model, doing so would require Alice to use a different Chunnel (e.g., `NqsbTLSChunnel`) to which the optimization in Listing 3.3 would not apply, thus guaranteeing safety.

However, Bertha cannot guarantee that optimizations improve performance, e.g., an application developer might have finer grained control when using separate RPC and TLS Chunnels (as is done in `conn1`), and thus in some cases separate Chunnels might result in better performance than using a single fused Chunnel. Thus, we can see that while Bertha's optimization framework provides an automatic way to correctly apply optimizations to an application, it requires that application developers analyze the impact of the optimization on application performance before applying them to the program.

3.4 Runtime Reconfigurability

Thus far, we have described how Bertha enables portable applications and allows developers to easily compose different Chunnels. We now look at

```
1  // Server code to accept connections from clients who can shard
2  // and ones where sharding is implemented in the middlebox.
3  let ser = SerializeChunnel::new(/*...*/);
4  let shrd = select!(MboxShard::new(/*..*/),
5          ClientShard::new(/*..*/));
6  let conn = tbm::make_stack!(ser, shrd).listen(addr);
7  // A client that implements client sharding.
8  let ser = SerializeChunnel::new(/*...*/);
9  let shrd = ClientShard::new(/*...*/);
10 let conn = tbm::make_stack!(ser, shrd).connect(server);
```

Listing 3.4: Bertha applications use `select!` to specify options for runtime reconfigurability.

how Bertha applications can provide runtime reconfigurability. A runtime-reconfigurable connection in Bertha is one for which the programmer specifies multiple possible Chunnel stacks, and Bertha chooses between these Chunnel stacks at runtime. As we explained in §3.1, choosing Chunnel stacks at runtime allows developers to create connections whose behavior can vary across clients. Adding support for runtime reconfigurability requires extending the interface described above and adding runtime logic.

Interface Changes.. The interface in §3.3.2 limits programmers to specifying a single Chunnel-stack per connection. We extend this interface by providing a `select!` (Listing 3.4 line 4) macro that the application programmer can use to specify a choice between Chunnels. On lines 1–5, Alice creates a connection that can perform shard selection either at a middlebox (`MboxShard`) or at the client (`ClientShard`). The connection `conn` (line 5) is thus associated with two different Chunnel stacks, and the runtime negotiation protocol we describe next chooses between them. We also show the code for a client that uses client sharding on lines 7 – 10; this client's stack is identical to that of one without runtime reconfigurability. In general, Bertha allows all endpoints in a connection (e.g., both the client and the server in this case) to use `select!`. In this case, the negotiation protocol selects a Chunnel stack that all endpoints can use. We refer to the resulting stack as the *datapath stack*.

3.4.1 Deciding the Datapath Stack

Bertha's negotiation protocol aims to select compatible Chunnel stacks when creating a connection. We say two Chunnels are compatible if data sent through one can be successfully received by the other and vice-versa. For example, two serialization Chunnels are compatible if data serialized by one can be deserialized by the other. Naturally, to check compatibility, Bertha needs to first collect (in a single endpoint) the set of Chunnel stacks specified by all endpoints. Thus, negotiating the datapath stack requires at least some communication. In what follows, we begin by describing how negotiation works for point-to-point connections that have two endpoints. We discuss how negotiation works when connecting among multiple endpoints (e.g., when using publish-subscribe Chunnels) after explaining our point-to-point mechanism.

As Figure 3.2 shows, this communication occurs in three phases. First, when an endpoint calls `connect` (in this section, we refer to this endpoint as the *client*), Bertha uses the specified network I/O library (§3.3.2) to establish a *base connection* to the other endpoint (which we refer to as the *server*). Our implementation supports a variety of network I/O libraries, including the sockets API and Shenango [102]. Note that while Bertha's negotiation protocol assumes reliability and ordering, we implement a simple

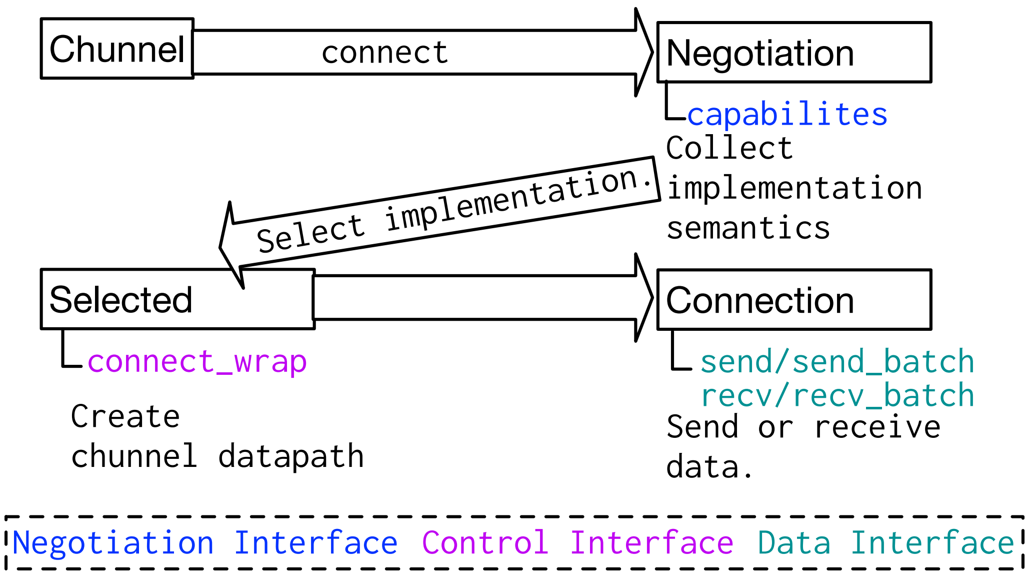


Figure 3.2: The Chunnel-implementation state machine: shows the sequence in which Bertha calls the interface functions for a single Chunnel-implementation.

reliability and ordering protocol as part of the negotiation logic. As a result, we do not require the network I/O library to implement TCP or other reliable transport protocols. Once the base connection has been established, the client uses this connection to send the server a message describing its Chunnel-stack. On receiving this message, the server checks whether the client has specified a compatible Chunnel stack to its own. If this check fails, the server returns an error to the client. Otherwise, the server selects a compatible Chunnel stack and sends the selected Chunnel stack to the client. The client and server then use this information to initialize a datapath stack by recursively calling `connect_wrap` as described in §3.3.1.

The protocol described requires that Bertha be able to check compatibility between pairs of Chunnels. A tempting approach to doing so would be to use static analysis to check compatibility between Chunnel implementations. However, Bertha does not constrain how developers write Chunnels, and checking compatibility is at least as hard as checking equivalence, and well known results in logic show that checking equivalence between programs is undecidable [122]. Therefore, in Bertha, we adopt a simpler approach where Chunnel developers describe the *relative* compatibility by providing an (opaque to Bertha) capability that implements a comparison function Bertha can use to check compatibility. To implement this mechanism, we require Chunnel developers to define a new capability type (or reuse one another Chunnel defines, to indicate compatibility with that Chunnel), and return an instance of this type when the Chunnel's `capability` function is called. A capability type must be serializable and must implement the `Capability` interface, which provides a function Bertha uses to check compatibility with another `Capability` instance. In practice, we found that in most cases, Chunnel capabilities can be encoded as a set of labels and that checking compatibility between them requires either checking that the sets are equal – we refer to this as exact-match below – or that there is a non-empty intersection between sets – we refer to this as composition.

Our design does not assume that capabilities are standardized. Instead, we only require that developers writing networked programs that communicate with each other (e.g., a client and a server) use the same capability types. This is similar to the current situation, where client developers must use the same RPC library as used by the server. Furthermore, the objective of Bertha capabilities is to offer an indication of relative compatibility: if OpenSSL provides an implementation of a TLS Chunnel, a later implementation based on `nqsb-TLS` would reuse the same capability types to indicate compatibility, and any future implementations could do the same.

Multi-Endpoint Connections.. Bertha supports connections with an arbitrarily large number of endpoints communicating over a single connection (e.g., multicast or publish-subscribe). Before designing a negotiation protocol for this scenario, we need to first determine how runtime reconfigurability works in a multi-endpoint setting. A point-to-point connection only exists for as long as the client and server are communicating and does not need to consider cases where endpoints join after the negotiation step has finished. However, this is not the case for connections with multiple endpoints. We can neither assume that all endpoints are known when a connection is first established nor can we require that all endpoints connect at the same time. As a result, we cannot use a mechanism where negotiation runs once when the connection is established and must provide mechanisms that allow (a) endpoints to recover a connection's datapath stack even if they did not participate in the negotiation round, and (b) when necessary, allow endpoints to trigger another round of negotiation and transition to a different datapath stack.

We implement a "rendezvous-based" negotiation protocol for the multi-endpoint case. We implement negotiation using a key-value store that can be accessed by all endpoints. The key-value store is also responsible for recording the negotiated datapath stack, thus allowing endpoints to recover the connection's datapath stack even when they do not participate in the negotiation. We require that the key-value store support serializable multi-key transactions. However, we do not impose other requirements, and we allow the use of both single-node key-value store (e.g., Redis) and replicated consensus based store (e.g., etcd) to be used for multi-endpoint negotiation. While the failure of this key-value store prevents new endpoints from joining a multi-endpoint connection, it has no impact on any endpoint that has already joined. Furthermore, the negotiation state is not shared across connections, and thus the key-value store can be easily sharded for scalability. While our use of an external-key value store has an impact on the time taken to establish a multi-endpoint connection, it is easy to see that any algorithm for multi-endpoint negotiation requires agreement and would thus impose similar performance costs.

A peer starts multi-party negotiation by connecting to the key-value store and proposing a Chunnel implementation stack, using compare-and-swap (e.g., via a transaction) to check for an existing stack. If this compare-and-swap operation succeeds, the peer can safely use the Chunnel implementation stack it proposed. Otherwise, the key-value store returns the Chunnel implementation stack already in place amongst the existing connection participants along with the number of participants in the connection. The peer then has two choices: use the existing semantics, or attempt to *upgrade* them. If the peer chooses to upgrade, it starts a two-phase commit process to transition its peers to its new preferred stack. Once the two-phase commit completes, all peers begin using the new stack. If peers refuse the new semantics, the new peer returns an error.[1] If a new endpoint arrives while an upgrade is in progress, it uses the new semantics and waits for the upgrade to commit before attempting to commit any upgrades. We demonstrate this process in §3.6.3.

3.5 Implementation

Next, we describe Bertha's implementation, starting with some optimizations that improve application performance. We evaluate our implementation's overheads and the impact of these optimizations in §3.6.1.

[1]Because all peers must accept the transition for it to commit, a faulty peer cannot force all connection participants to switch stacks.

3.5.1 Optimizations

Zero-RTT Negotiation. As we note in §3.4, runtime-reconfigurability requires agreement, and hence endpoints must communicate before they can transfer data. The point-to-point protocol described in §3.4 completes in one RTT. While this RTT can also be simultaneously used by other protocols, e.g., TLS, that need a setup phase, it does lead to a modest increase in connection establishment time. To address this, the Bertha implements includes an additional optimization that allows connections to be reestablished without additional negotiation, thus reducing overheads for applications that use many short-lived connections. Our approach to doing so is inspired by QUIC's zero-RTT [76] connection establishment.

Zero-RTT negotiation requires the client to remember the datapath stack used by previous connections and re-use it when reconnecting. We modify the `connect` call to do so by having the client send the server a zero-RTT negotiation message when it knows of a previously negotiated stack and then instantiating that stack. The client can send data once the stack has been instantiated. When the server receives a zero-RTT negotiation message, it checks if the previously negotiated stack can still be used. If so, the server re-initiates the stack and uses it to process all subsequent data. If, however, the server cannot use the stack for some reason, it sends the client a message indicating that negotiation has failed and proposing a new stack. If Bertha on the client receives such a message indicating negotiation failure, it tears down the existing stack and instantiates the stack included in the failure message.

Datapath Batching.. Bertha allows Chunnel writers to optionally provide functions for sending and receiving batches of data. To do so, Chunnel writers define `send_batch` and `recv_batch` functions as a part of the Chunnel datapath. The default implementations of these functions, used when a Chunnel-implementation does not provide an implementation, simply calls `send` (or `recv`) for each message in the batch. Bertha also provides an additional connection type, `nagling_connection`, that automatically batches `send` and `recv` calls using Nagle's algorithm. This makes it easier for application to benefit from batching, and we evaluate this approach in §3.6.3.

3.5.2 Bertha Structure

As we stated previously, we have implemented Bertha in Rust. While our design uses several Rust features, including traits and procedural macros, we believe our ideas are more general and can be implemented using similar features and metaprogramming support in other languages. For example, we believe C++ templates, Racket macros, and Go's generate tool would allow us to implement all of the features we have described in those languages.

We implement Chunnel stacks with a structure similar to Lisp's `cons`, and we represent choices between Chunnel implementations using a `Select` struct that contains both branches. Since it represents multiple options, the `Select` type does not implement `connect_wrap`; instead, Bertha picks one of the `Select`'s options (using the negotiation process) and replaces it in the Chunnel stack with an enum variant representing the choice. The enum delegates the `connect_wrap` implementation to the choice it represents.

To expose negotiation functionality to applications, Bertha needs only a method call for the two-endpoint case. Because multi-endpoint negotiation requires maintaining agreement, Bertha provides a datapath agent that listens for updates concurrently with any calls to `send` and `recv` on the application's connection and dynamically transitions the connection stack appropriately.

Bertha provides a `Subst` procedural macro that helps developers implement stack passes using a mini-DSL. To use this macro, developers define a Rust trait with a generic output type. Then, developers specify a substitution over arbitrary Chunnel stacks. Bertha interprets this mini-DSL to generate code that implements the trait for various Chunnel stack types. Application developers can then use the optimizations by calling the trait method on their Chunnel stack. If the stack doesn't match the optimization pattern, the optimization method will be a no-op. Using the `Subst` macro is optional; developers can implement their stack pass trait manually over Chunnel stack types.

We implement Bertha's core libraries (and tests) in ~5,300 lines of Rust, including the three interfaces described above (together ~1,300 lines) and the negotiation protocol (~4,000 lines). We further implemented Chunnels for the applications described in §3.6 in ~3,000 lines of code, including serialization, reliability, and ordering and compatibility wrappers that allow Bertha applications to use UDP sockets, Unix-domain sockets, and Shenango [102] as base connections. Finally, we implement Chunnels specific to our application case studies, which we describe in §3.6.

3.6 Evaluation

Next, we evaluate the performance, generality, and utility of Bertha. We first show that Bertha is not a throughput bottleneck and measure the time Bertha takes to establish a connection. We then demonstrate both the generality of the Chunnel abstraction and Bertha's utility by discussing three applications we have implemented using Bertha. We implemented a variety of Chunnels to enable these applications, including a microservice

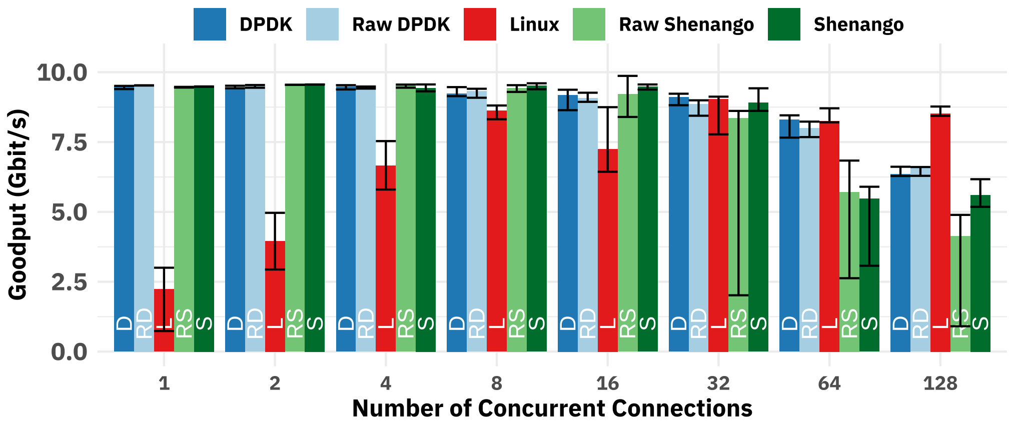


Figure 3.3: Bertha achieves the performance the underlying datapath provides. This microbenchmark used a 10Gbit/s NIC. "Raw" represents that datapath's performance without Bertha. Columns show p50 and errorbars show p25 and p75.

Chunnel that allows colocated microservices to communicate using a fast path (§3.6.2), several publish-subscribe Chunnels that expose third-party publish-subscribe services (§3.6.3), and several Chunnels that implement sharding and load-balancing across applications. With these examples, we show that Bertha enables portability, simplifies composition, and enables runtime reconfigurability.

Unless otherwise stated, we ran all evaluation on servers running Linux kernel 5.4.0. All servers were on the same rack and used 4-core 2.80GHz Intel Xeon E5-1410 CPUs and Mellanox CX-3 Pro NICs with OFED driver version 5.0.1, and were connected with 10 Gbit/s links.

3.6.1 Bertha Overheads

Throughput.. We measure Bertha's impact on network throughput using a file transfer application. In this application, the client establishes a connection to the server and sends the server a short request. The server sends a 50MB file in response, and we measure "goodput:" the time taken to receive this file. We implement this application using a custom DPDK datapath ("Raw DPDK"), Shenango ("Raw Shenango")[2], and Bertha. We run the Bertha version of the application with three network I/O libraries: Shenango, DPDK, and the Linux's socket interface ("Kernel"). The Bertha application transfers data over a connection whose Chunnel stack is comprised of a single no-op Chunnel. Our no-op Chunnel neither changes the data being sent (or received) nor alters routing, thus allowing us to directly measure Bertha's overheads.

We show the throughput achieved across these configurations in Figure 3.3. Bertha has negligible impact on throughput, and this overhead does not increase as a function of the number of concurrent connections. This benchmark also demonstrates Bertha's support for application portability: changing the Bertha application to use Linux sockets instead of DPDK required changing a single line of code. Adopting Shenango for this benchmark required more extensive changes, because Shenango's runtime must be aware of any blocking calls applications make. These changes lie outside of the communication features that we consider in our work.

Connection Establishment.. Next, we evaluate how long it takes for Bertha to establish a connection. We use a point-to-point echo application and measure time taken to create a connection, send a short 8-Byte message, and receive the echoed response. In Figure 3.4 we show the distribution of

[2]We used Caladan [44], an updated version of Shenango.

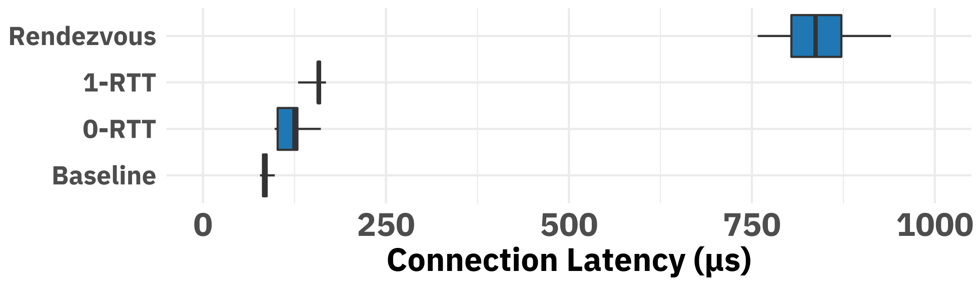


Figure 3.4: Time to establish a connection and send and receive one message. “Baseline” uses Linux UDP. Boxplots in this paper show (p5, p25, p50, p75, p95).

response times across 1,000 trials when using the full negotiation protocol (1-RTT) and when using the 0-RTT negotiation protocol described in §3.5. We compare this to a “Baseline” implementation using UDP sockets. While both negotiation protocols increase the time taken to establish a connection, the increase with 0-RTT negotiation is modest. Finally, we also measure the cost of our rendezvous-based multi-endpoint negotiation protocol (“Rendezvous”). While it adds significantly to connection establishment time, as we show later it is a small fraction of pub-sub services’ latency (§3.6.3).

3.6.2 Microservice Communication

Slim [144] and similar efforts have shown that there are significant performance benefits to using a network overlay that handles traffic between microservices co-located on the same server different than it does for traffic going between servers. Slim showed that traffic between colocated microservices can be isolated without the cost of the network stack and virtual switch. We implemented a `MaybeLocalChunnel` that provides Slim-like functionality in Bertha. When establishing a connection, this Chunnel consults a machine-local directory to check if the other endpoint is colocated on the same server. If so, `MaybeLocalChunnel` uses a Unix-domain socket (and otherwise a normal connection). Our implementation of the `MaybeLocalChunnel` comprises ~500 lines of Rust code, and the machine-local directory is another ~300 lines.

We also implemented a TLS Chunnel to support microservices, because most microservice applications use TLS to authenticate endpoints and encrypt traffic. Our `TLSChunnel` uses Ghostunnel [123], a TLS proxy, for this purpose, and is written in ~700 lines. Note that Ghostunnel runs as an external process, which is spawned when the first connection using this Chunnel is created, adding to connection establishment latency for this connection. Subsequent connections reuse this process, and as a result those connections do not incur additional latency.

Finally, we observe that applications use TLS to protect against network attacks. However, these attacks are not a concern when sending data between microservices colocated on the same server. We thus implement a `FusedTLSMaybeLocalChunnel` that uses TLS when sending data between microservices on different servers, and uses `MaybeLocalChunnel`'s shared-memory channel without encryption for colocated microservice. This Chunnel comprises of 250 lines of Rust code, and simply calls into the above Chunnels. We also wrote an optimization pass (§3.3.3) in 2 lines in the `Subst` mini-DSL to replace occurrences of `TLSChunnel` composed with `MaybeLocalChunnel` with this fused Chunnel.

We used the three Chunnels and optimization pass to implement a simple echo application. Both the client and the server in this case are colocated on the same server and run in Docker containers. In Figure 3.5, we show request latency by issuing $20,000$ requests across 200 connections. We show latency figures for three cases: (a) "Bertha," where we compose

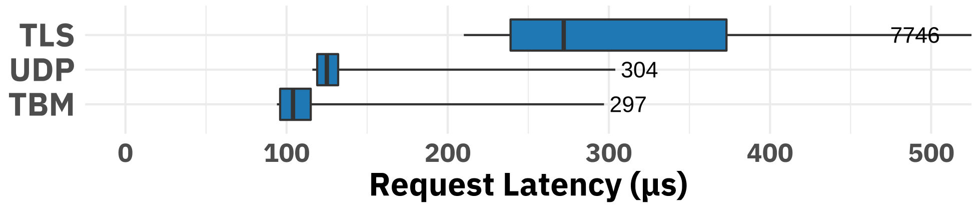


Figure 3.5: When a microservice is local, we can avoid the overhead of TLS using the fastpath Chunnel (10 requests per connection).

a `MaybeLocalChunnel` with a `TLSChunnel` and use an optimization pass to replace these with a `FusedTLSMaybeLocalChunnel`; (b) "UDP," where we use neither a `MaybeLocalChunnel` nor a `TLSChunnel`; and (c) "TLS," where we compose a `MaybeLocalChunnel` with a `TLSChunnel`, but do not use the optimization pass. The 'TBM' case, which uses both Chunnels and the optimization pass, shows the lowest request latency. 'TLS,' which also uses both Chunnels but omits the optimization, shows the worst, demonstrating the benefit of Bertha's optimization passes. 'TLS''s high tail latency is due to it spawning Ghostunnel when establishing a connection. Finally, the difference in request latency between 'UDP' and 'TBM' shows the benefit of a Slim-like approach. Switching between these two settings requires changing a single line of code, thus demonstrating the benefit of Bertha's unified interfaces.

3.6.3 Publish-Subscribe Message Queues

Many cloud applications use message queues and publish-subscribe services for elastic scaling. These services allow replicated servers to pull messages when they have available processing capacity[3], simplifying load-balancing

[3]Implementations can also support 'push-based' delivery, where servers are notified when messages are available.

Provider	Ordering	Multicast
AWS SQS	Set at queue creation	Compose with SNS
GCP PubSub	Set at subscription creation	Per-Subscription
Azure Queues	Unsupported	Unsupported
Kafka	Always ordered	Per-Subscription

Table 3.2: Example of semantic possibilities with message queues. We describe the change required to transition from a connection with best effort ordering and message-spraying delivery to one with ordering or multicast semantics.

and task distribution.

Publish-subscribe services are available from most cloud providers, including Amazon (AWS SQS [4]), Google (GCP PubSub [48]), and Azure (Azure Storage Queues [88] and Service Bus [89][4]). Kafka [75] and other open source projects also implement these services. However, as we show in Table 3.2, each implementation offers different ordering and delivery semantics. In terms of ordering, AWS SQS and GCP PubSub support in-order message delivery, Azure Storage Queues does not guarantee any message ordering, and Kafka always guarantees message ordering. Similarly, in terms of delivery semantics (the Multicast column in Table 3.2) GCP PubSub and Kafka allow application developers to create topics or subscriptions where multiple receivers can receive the same message, but AWS SQS requires the use of a separate service (AWS SNS [3]), and Azure Queues does not natively support multicast. Overall, when using these services application

[4]At present, Azure Service Bus does not provide a Rust API, and thus we do not consider further it in this section.

developers need to both use provider specific APIs and account for these semantic differences. These API and semantic differences impede portability across cloud providers.

Furthermore, prices and pricing schemes for these services can vary significantly. For example, as of early 2022 AWS SQS provides 1 million free 64KB messages per month and charges \$0.40 per million (\$0.50 per million) for subsequent unordered (ordered) messages. Google's GCP PubSub provides up to 10GB of free messages per month with a minimum message billing size of $1,000$ bytes, and then \$40/TB. Azure Storage Queues [88] charges \$0.004/$10,000$ messages. Thus, deployment costs can vary depending on the service and workload. The lack of portability can thus result in higher application deployment costs.

To demonstrate that the Chunnel abstraction can improve application portability, we created a Chunnel for each of the four services (AWS SQS, GCP PubSub, Azure Storage Queues and Kafka) listed above. These Chunnels took between 250 and 400 lines of code each. Of course, in the general case, Chunnels (whose logic runs on a single machine) cannot abstract away semantic differences between services. However, in the special case where there is a single receiver, we can have the receiver implement ordering semantics. To demonstrate this we also implemented an ordering Chunnel in ~400 lines of Rust, and an at-most-once delivery Chunnel in ~200 lines of Rust.

We evaluated the Chunnels described above by creating an application that composes them with a simple serialization Chunnel implemented in 200 lines using Bincode [103] and a Base64 encoder. We use this application to measure message latencies across these services in 13 scenarios (Figure 3.6). In these evaluations, we configured both a virtual machine and the cloud provider's message service in one of the cloud provider's dat-

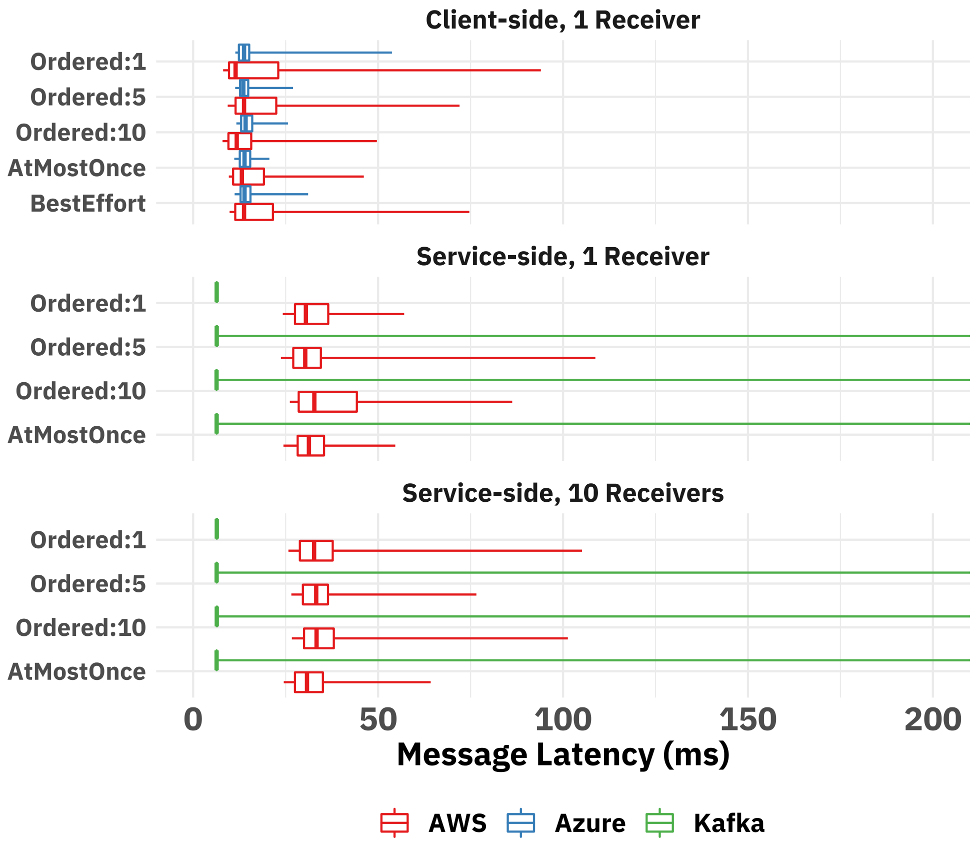


Figure 3.6: Message queue providers have different performance characteristics and semantics. “Ordered:n” means ordering among n groups.

```
tbm::make_stack!(SerializeBase64,
  tbm::select!(HostOrderedSqs, ServiceOrderedSqs,
    tbm::policy::NumPeersLessThanEq(2) => Left)))
  .connect(topic);
```

Listing 3.5: A multi-endpoint Chunnel stack that specifies when to switch between two Chunnel options.

acenters. We installed Kafka on a local server, and ran the client on another server in the same rack. We measured latency by configuring the application to send and receive 100 messages using the publish-subscribe service with message interarrival time of 75ms, and split messages evenly across ordering groups when using multiple groups. Figure 3.6 shows latencies for AWS SQS, Azure Storage Queues and Kafka. We do not show GCP PubSub latencies, because we ran into a known issue where GCP has high latencies when using a small number of messages [15]. We similarly believe that Kafka's high-tail latencies are due to our use of its default configuration. The latencies we measured for the Bertha application were identical to what we measured when using an application that directly called into the cloud provider libraries. Furthermore, changing between these 14 scenarios required changes to a single line of code: we merely had to change Chunnel types or configuration. This experiment demonstrates two things: first, using Bertha to access cloud services does not carry a latency penalty, and second, Bertha's unified interface improves application portability.

Multi-Endpoint Runtime Re-configuration.. While developing these Chunnels, we observed that Bertha's ordering Chunnel provides lower latency than AWS SQS and GCP PubSub's ordered subscriptions. However, as we mentioned previously, Bertha's ordering Chunnel can only be used

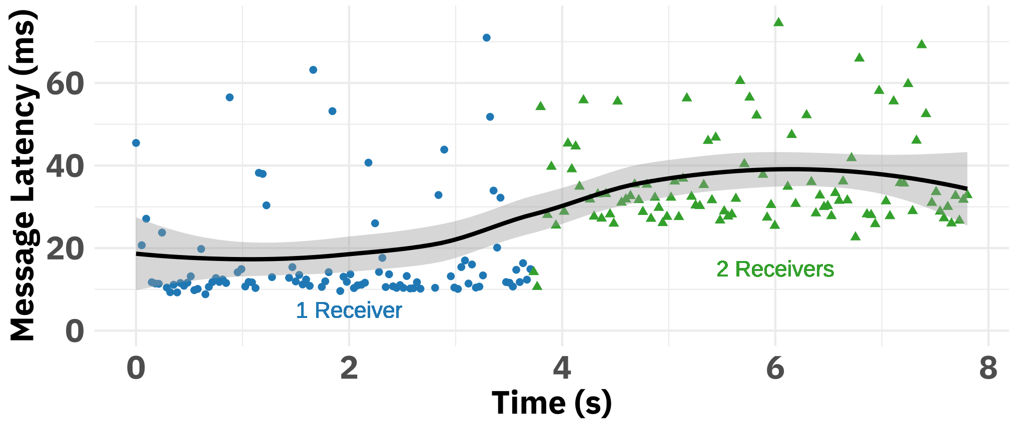
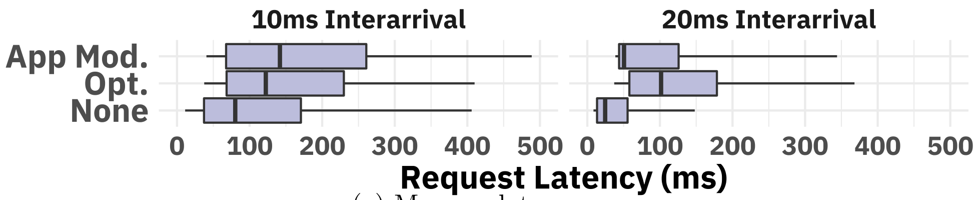


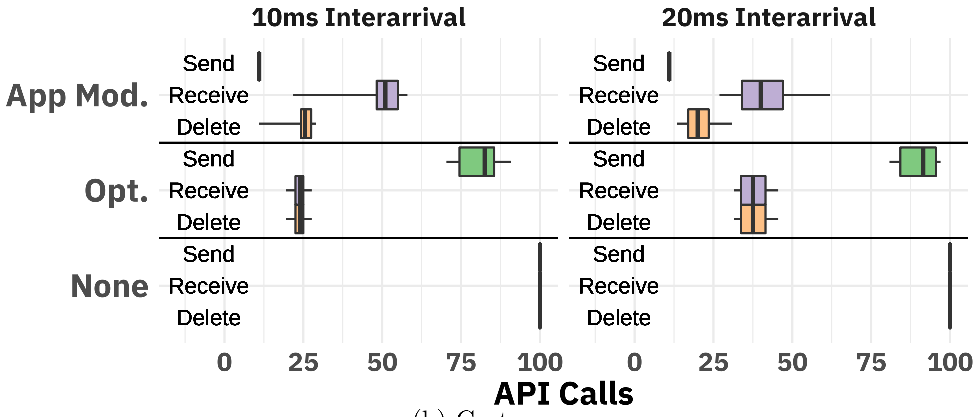
Figure 3.7: Runtime reconfiguration allows switching datapath stacks. Note that the first few requests after the second receiver arrives still use the original stack while the transition commits.

safely when a single receiver is active. We took advantage of Bertha's runtime reconfigurability (§3.4) to implement an application that can switch between receiver-side ordering and service ordering depending on the number of receivers. We show a snippet of this application in Listing 3.5. Note that this snippet uses a more general version of the `select!` macro we described in §3.4: it accepts a predicate (`tbm::policy::NumPeersLessThanEq(2)`) that breaks ties between compatible Chunnel stacks, and (in this case) prefers receiver-side ordering. In Figure 3.7, we send 100 messages with an inter-arrival time of 25ms, then start a second receiver and send another 100 messages. We use 5 ordering groups and AWS SQS. We can see that the application uses receive-side ordering and observes lower message latencies when a single receiver is present, and then safely transitions to the higher-latency but safer service ordering when the second receiver arrives.

Batching.. Many cloud-provider publish-subscribe services allow develop-



(a) Message latency.



(b) Costs.

Figure 3.8: Automatic batching optimization ("Opt.") can reduce API costs with comparable latency. The Opt. configuration Nagles messages, so shorter message interarrivals allow more cost savings for sends.

ers to send batches of messages using a single call, but charge per-request. As a result, batching can significantly reduce the cost of using these services; e.g., AWS SQS allows up to 10 messages per call, so batching SQS API calls can reduce costs up to 10x.

As we described in §3.5.1, Bertha provides a `nagling_connection` that automatically batches an application's send and receive calls. We evaluate this in Figure 3.8 by measuring (across 10 runs) the number of API calls needed to transmit and receive 100 unordered SQS messages and the corre-

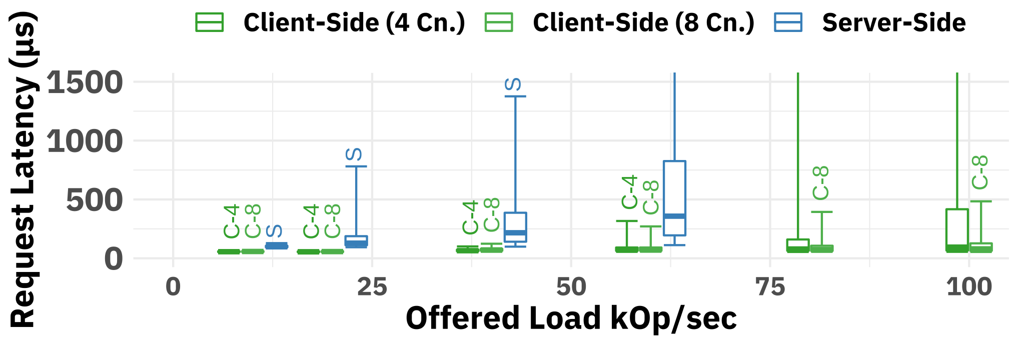


Figure 3.9: Client sharding scales better with lower latency, and can take advantage of more concurrency at the client without burdening the server implementation with more connections.

sponding message latencies. "None" shows the baseline no-batching version: it achieves the lowest latency, but issues one send, one receive, and one delete API call per message. On the other extreme, "App Modification" uses the fewest requests because the application developer groups messages into batches and the application uses exactly 11 send API calls. The "Optimization" configuration, uses a `nagling_connection` and achieves the same receive-side API cost as the "App Modification" configuration, but because it relies on "Nagling" it has a larger number of send batches. The number of batches used in this case depends on the workload: with 10ms interarrival times, nagling uses 82 calls, but uses 92 when the interarrival time increases to 20ms. Both "App Modification" and "Optimization" achieve latencies within 2x of the baseline case: 77 % and 52 % higher, respectively.

3.6.4 Sharding and Load Balancing

Another common component of modern applications is application-aware, or "L7" load balancing, e.g., Amazon's "Application Load Balancing" (ALB),

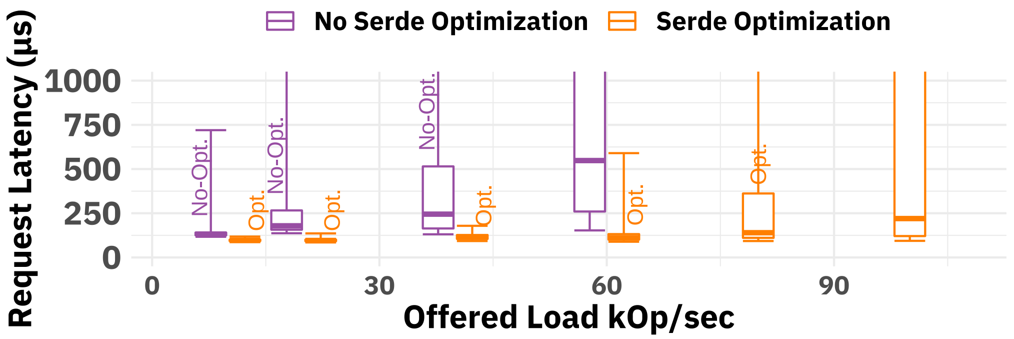


Figure 3.10: Here, shards are on separate machines and the sharder forwards requests over the network. Applying an optimization that skips serialization and deserialization helps server-sharding to scale better.

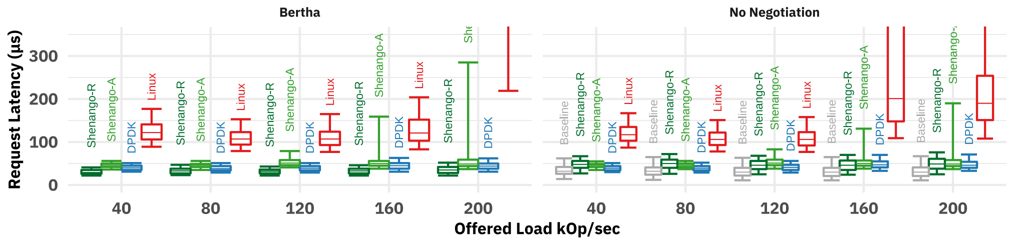


Figure 3.11: Performance across Bertha datapaths with and without negotiation and Chunnels. “Shenango-A” runs Shenango’s runtime in a thread and communicates with it via a memory channel, while “Shenango-R” re-implements Bertha’s features within the Shenango runtime. “Baseline,” also implemented inside the Shenango runtime, removes Chunnels in addition to negotiation.

Azure's "Application Gateway", or GCP's "Internal HTTP(S) Load Balancing." These services can shard HTTP(s) requests based on pre-configured rules on URL parameters, and can auto-scale services. Other application aware load balancers can read more complex formats, including specialized formats used by key-value stores like Memcached and Redis.

To show how load balancing and sharding work with Bertha, we implemented a server and client sharding Chunnel designed for use with a key-value store. Both Chunnels determine the shard where a request should be routed by hashing the first 4 bytes of the key (using the FNV algorithm [42]) and modding it by the number of shards. The server sharding Chunnel, which is implemented in ~450 lines of Rust, reads the request and determines the shard on the server side. Once it has determined the shard it forwards the request to a backend either using a channel (if the backend is on the same server) or via UDP if the shard is remote. The client sharding Chunnel determines the shard before forwarding the request, and then sends the request to the appropriate shard.

We evaluate the Chunnels in this section using a simple key-value store that we built using a publicly available concurrent hashmap implementation that uses binned locking (`dashmap` [1]). Our evaluations are run on a cluster with 6 shards, all of which run as threads on the same server. We evaluate performance using the YCSB benchmark: we first run a warm up phase and loading phase that issues $12,000$ PUT requests, then we use 5 clients (each running on its own server) to issue requests according to the "Workload B" request distribution. Note that YCSB messages are 132B in size. Each client issues $277,000$ requests. We control the request rate to meet our offered load, and each client splits requests across 4 connections. To ensure that the offered load remains consistent, we terminate the experiment after any connection sends its last request.

Runtime Reconfiguration for Client Sharding. As we discussed previously in §3.4, Bertha's runtime reconfiguarability lets applications choose between client side and server side sharding when establishing a connection. In Figure 3.9 we show that client sharding: (a) reduces server overheads and thus achieves higher throughput: when using server side sharding, the key-value store cannot keep up with request rates of 75 or 100 kOps/second, but can do so with client side sharding; (b) lowers response latency; and (c) allows clients to scale to more connections. Additionally, the negotiation process ensures that client sharding is safe, because the sharding capabilities used encode the policy, and thus we know that the client and server agree on the sharding policy.

Optimizations Bertha Enables. We observe that a connection's datapath stack specifies the wire format of the data both the client is sending and that the shard is expecting to receive, and we know these will match because of negotiation. Therefore, we implement an optimization that replaces `ShardChunnel` with `ShardChunnelRaw`, which takes a generic parameter corresponding to the key's byte offset in the packet buffer, evaluates the shard function at that offset, and forwards to the correct shard without needing to deserialize the buffer. Importantly, without Bertha this stack pass would be unsafe; using negotiation ties `ShardChunnelRaw`'s byte offset to `SerializeChunnel`'s implementation and so ensures that using a fixed byte offset is correct. We can see the impact in Figure 3.10. Here, we use 2 remote machines in the same rack as shards and disable client-side sharding. This configuration is similar to that of Amazon ALB, Azure Application Gateway, or GCP HTTP Load Balancing, though the approach we describe can generalize to any wire format with fixed, deterministic byte offsets for fields. For example, rather than specializing to an individual protocol (as is common [46, 65, 74, 127]), implementations on accelerator hardware could use this approach for generality.

Portability and Bertha Overheads.. Finally, we use the key-value store to demonstrate Bertha's ability to switch between network I/O libraries and to evaluate the cost of Bertha's abstractions. In Figure 3.11, we show the application's performance with and without Bertha's features on four underlying datapath configurations. Overall, Bertha achieves comparable performance as the underlying datapath it uses. Note, that each of the datapath offers different features, and imposes different deployment requirements. In all cases, Bertha inherits these features and requirements. Importantly, switching between these datapaths requires changing only a single-line of code, making it easier for developers and users to switch between these approaches.

As we discussed earlier, porting an application to Shenango can require extensive changes in the runtime API. We explored two approaches to porting the key-value store to Shenango: (a) "Shenango-A", where the application communicates with the Shenango runtime over a memory channel, and thus switching to the Shenango-A datapath remains a one-line change, similar to Linux or DPDK; and (b) "Shenango-R", where we re-implemented name's features and all Chunnels needed for this application within the Shenango runtime, using its concurrency APIs. We can see from Figure 3.11 that the "Shenango-R" application can achieve lower latencies than Shenango-A or DPDK, but at the cost of portability.

We also re-implemented the same application in two configurations without Bertha's features. "No Negotiation" configurations remove negotiation from the application (thus removing the ability for runtime reconfiguration), and "Baseline" further removes Chunnels, in-lining the entire application.

In all these cases, removing Bertha features resulted in only minor performance gains; i.e., Bertha applications achieve performance determined by their underlying datapath rather than Bertha itself.

3.7 Conclusion

We have described Bertha, a runtime-reconfigurable networking stack that allows developers to compose libraries implementing connection functionality into individual connections their applications can use. Bertha's key features are the Chunnel abstraction, which enables composability, portability, and optimizations, and negotiation, which further enables runtime reconfiguration.

4 CCP

4.1 Introduction

At its core, a congestion control protocol determines when each segment of data must be sent. Because a natural place to make this decision is within the transport layer, congestion control today is tightly woven into kernel TCP software and runs independently for each TCP connection.

This design has three shortcomings. First, many modern proposals use techniques such as Bayesian forecasts (Sprout [136]), offline or online learning (Remy [135], PCC [30], PCC-Vivace [31], Indigo [139]), or signal processing with Fourier transforms (Nimbus [51]) that are difficult, if not impossible, to implement in a kernel lacking useful libraries for the required calculations. For example, computing the cube root function in Linux's Cubic implementation requires using a table lookup and a Newton-Raphson iteration instead of a simple function call. Moreover, to meet tight performance constraints, in-kernel congestion control methods have largely been restricted to simple window or rate arithmetic.

Second, the kernel TCP stack is but one example of a *datapath*, the term we use for any module that provides data transmission and reception interfaces between higher-layer applications and lower-layer network hardware. Recently, new datapaths have emerged, including user-space protocols atop UDP (e.g., QUIC [76], WebRTC [66], Mosh [134]), kernel-bypass methods (e.g., mTCP/DPDK [32,63,114]), RDMA [143], multi-path TCP (MPTCP) [137], and specialized Network Interface Cards ("SmartNICs" [96]). This trend suggests that future applications will use datapaths different from traditional kernel-supported TCP connections.

New datapaths offer limited choices for congestion control because implementing these algorithms correctly takes considerable time and effort. We believe this significantly hinders experimentation and innovation both in the datapaths and the congestion control algorithms running over them. For instance, the set of available algorithms in mTCP [63], a TCP implementation on DPDK, is limited to a variant of Reno. QUIC, despite Google's imposing engineering resources, does not have implementations of several algorithms that have existed in the Linux kernel for many years. We expect this situation to worsen with the emergence of new hardware accelerators and programmable network interface cards (NICs) because high-speed hardware designers tend to forego programming convenience for performance.

Third, tying congestion control tightly to the datapath makes it hard to provide new capabilities, such as aggregating congestion information across flows that share common bottlenecks, as proposed in the Congestion Manager project [8].

If, instead, the datapath encapsulated the information available to it about *congestion signals* like packet round-trip times (RTT), receptions, losses, ECN, etc., and periodically provided this information to an off-datapath module, then congestion control algorithms could run in the context of that module. By exposing an analogous interface to control transmission parameters such as the window size, pacing rate, and transmission pattern, the datapath could transmit data according to the policies specified by the off-datapath congestion control algorithm. Of course, the datapath must be modified to expose such an interface, but this effort needs to be undertaken only once for each datapath.

We use the term *Congestion Control Plane (CCP)* to refer to this off-datapath module. Running congestion control in the CCP offers the following benefits:

1. **Write-once, run-anywhere:** One can write a congestion control algorithm once and run it on any datapath that supports the specified interface. We describe several algorithms running on three datapaths: the Linux kernel, mTCP/DPDK, and QUIC, and show algorithms running for the first time on certain datapaths (e.g., Cubic on mTCP/DPDK and Copa on QUIC).

2. **Higher pace of development:** With good abstractions, a congestion control designer can focus on the algorithmic essentials without worrying about the details and data structures of the datapath. The resulting code is easier to read and maintain. In our implementation, congestion control algorithms in CCP are written in Rust or Python and run in user space.

3. **New capabilities:** CCP makes it easier to provide new capabilities, such as aggregate control of multiple flows [8], and algorithms that require sophisticated computation (e.g., signal processing, machine learning, etc.) running in user-space programming environments.

This paper's contributions include:

- An event-driven language to specify congestion control algorithms. Algorithm developers specify congestion control behavior using combinations of events and conditions, such as the receipt of an ACK or a loss event, along with corresponding handlers to perform simple computations directly in the datapath (e.g., increment the window) or defer complex logic to a user-space component. We show how to implement several recently proposed algorithms and also congestion-manager aggregation.

- A specification of datapath responsibilities. These include congestion signals that a datapath should maintain (Table 4.2), as well as a simple framework to execute directives from a CCP program. This design enables "write-once, run-anywhere" protocols.
- An evaluation of the fidelity of CCP relative to in-kernel implementations under a variety of link conditions. Our CCP implementation matches the performance of Linux kernel implementations at only a small overhead (5% higher CPU utilization in the worst case).

4.2 Related Work

The Congestion Manager (CM [8]) proposed a kernel module to separate congestion control from individual flows. CM provides an API for flows to govern their transmissions and a plan to aggregate congestion information across flows believed to share a bottleneck. The CM API requires a flow to inform the CM whenever it wanted to send data; at some point in the future, the CM will issue a callback to the flow granting it permission to send a specified amount of data. Unlike CCP, the CM architecture does not support non-kernel datapaths or allow custom congestion control algorithms. Further, the performance of CM is sub-optimal if the CM and the datapath are in different address spaces, since each permission grant (typically on each new ACK) requires a context switch which reduces throughput and increases latency. We show in §4.7.3 that CCP can support the aggregate congestion control capabilities of the CM architecture.

eBPF [35] allows developers to define programs that can be safely executed in the Linux kernel. These programs can be compiled just-in-time (JIT) and attached to kernel functions for debugging. TCP BPF [13] is an extension to eBPF that allows matching on flow metadata (i.e., 4-tuple) to customize TCP connection settings, such as the TCP buffer size or SYN RTO. In the kernel datapath, it may be possible for CCP to use the JIT features of eBPF to gather measurements, but not (yet) to set rates and congestion windows. Exploring the possibility of TCP control entrypoints for eBPF, and an implementation of a Linux kernel datapath for CCP based on such control, is left for future work.

Linux includes a pluggable TCP API [25], which exposes various statistics for every connection, including delay, rates averaged over the past RTT, ECN information, timeouts, and packet loss. icTCP [53] is a modified TCP stack in the Linux kernel that allows user-space programs to modify specific TCP-related variables, such as the congestion window, slow start threshold, receive window size, and retransmission timeout. QUIC [76] also offers pluggable congestion control. We use these Linux and QUIC pluggable APIs to implement datapath support for CCP. CCP's API draws from them, but emphasizes asynchronous control over datapaths.

HotCocoa [5] introduces a domain specific language to allow developers to compile congestion control algorithms directly into programmable NICs to increase efficiency in packet processing. In contrast, CCP allows developers to write algorithms in user-space with the full benefit of libraries and conveniences such as floating point operations (e.g., for Fourier transforms).

Structured Streams (SST [41]) proposed a datapath that prevents headof-line blocking among packets of applications by managing the transport streams between a given pair of hosts and applying a hereditary structure on the streams. Unlike SST, CCP does not manage the contents of the underlying transport stream: CCP enables deciding when a packet is transmitted, not which packet. We view SST and CCP as complementary architectures which can be combined to provide composable benefits.

Finally, there is a wide range of previous literature on moving kernel functionality into user-space. Arrakis [107] is system that facilitates kernel-bypass networking for applications via SR-IOV. IX [10] is a dataplane operating system that separates the management functionality of the kernel from packet processing. Alpine [36] moves all of TCP and IP into user-space. Whereas these systems use hardware virtualization to allow applications to have finer grained control over their networking resources, CCP exposes only congestion control information to user-space. Moreover, CCP is also agnostic to the datapath; datapaths for library operating systems could be CCP datapaths.

4.3 CCP Design Principles

To enable rich new congestion control algorithms on datapaths, CCP must provide a low-barrier programming environment and access to libraries (e.g., for optimization, machine learning, etc.). Further, new algorithms should also achieve high performance running at tens of Gbit/s per connection with small packet delays in the datapath.

4.3.1 Isolating Algorithms from the Datapath

Should congestion control algorithms run in the same address space as the datapath? There are conflicting factors to consider:

Safety. Supporting experimentation with algorithms and the possibility of including user-space code means that programs implementing congestion control algorithms should be considered untrusted. If algorithms and the datapath are in the same address space, bugs in algorithm or library code could cause datapath crashes or create vulnerabilities leading to privilege escalations in the kernel datapath.

Flexibility. Placing congestion control functionality outside the datapath provides more flexibility. For example, we anticipate future use cases of the CCP architecture where a congestion control algorithm may run on a machine different from the sender, enabling control policies across groups of hosts.

Performance. On the other hand, congestion control algorithms can access the datapath's congestion measurements with low delays and high throughput if the two reside in the same address space.

Our design restructures congestion control algorithms into two components in separate address spaces: an off-datapath *CCP agent* and a component that executes in the datapath itself. The CCP agent provides a flexible execution environment in user space for congestion control algorithms, by receiving congestion signals from the datapath and invoking the algorithm code on these signals. Algorithm developers have full access to the user-space programming environment, including tools and libraries. The datapath component is responsible for processing feedback (e.g., TCP or QUIC ACKs, packet delays, etc.) from the network and the receiver, and

Implementation	Reporting Interval	Mean Throughput
Kernel	-	43 Gbit/s
CCP	Per ACK	29 Gbit/s
CCP	Per 10 ms	41 Gbit/s

Table 4.1: Single-flow throughput for different reporting intervals between the Linux kernel and CCP user-space, compared to kernel TCP throughput. Per-ACK feedback (0 μs interval) reduces throughput by 32% while using a 10 ms reporting interval achieves almost identical throughput to the kernel. Results as the number of flows increases are in §4.6.2.

providing congestion signals to the algorithms. Further, the datapath component provides interfaces for algorithms to set congestion windows and pacing rates.

An alternative design would be to run both the algorithm and the datapath in the same address space, but with fault isolation techniques [22, 38, 82, 111, 126, 132, 140]. However, this approach comes with significantly increased CPU utilization (e.g., 2× [22, 82, 111, 126, 132], resulting from tracing and run-time checks), a restrictive development environment [140], or changes to development tools such as the compiler [38, 132]. These performance and usability impediments, in our view, significantly diminish the benefits of running congestion control algorithms and the datapath in one address space.

4.3.2 Decoupling Congestion Control from the ACK Clock

Typical congestion control implementations in the Linux kernel are coupled to the so-called "ACK-clock," i.e., algorithm functionality is invoked upon receiving a packet acknowledgment in the networking stack. In contrast, with CCP, algorithms operate on summaries of network observations obtained over multiple measurements gathered in the datapath. Users program the datapath to gather these summaries using a safe domain-specific language (§4.3.3).

This decoupling of algorithm logic from the ACK clock provides two benefits.

First, users can develop congestion control algorithms free from the strict time restrictions rooted in the inter-arrival time of packet acknowledgments—a useful feature, especially at high link rates. Hence, it is possible to build algorithms that perform complex computations and yet achieve high throughput.

Second, the ability to provide congestion feedback less frequently than per-ACK can significantly reduce the overhead of datapath-CCP communication. Table 4.1 shows that for a single saturating `iperf` connection over a loopback interface, Linux kernel TCP on a server machine with four 2.8-Ghz cores achieves 45 Gbit/s running Reno. In comparison, per-ACK reporting from the kernel to the CCP agent achieves only 68% of the kernel's throughput. By increasing the time between reports sent to the slow path to 10 ms (see the "per 10 ms" row), our implementation of Reno in CCP achieves close to the kernel's throughput.

Given that CCP algorithms operate over measurements supplied only infrequently, a key question is how best to summarize congestion signals within the datapath so algorithms can achieve high fidelity compared to a traditional in-datapath implementation. Indeed, in §4.6.1 we show that reporting on an RTT time-scale does not affect the fidelity of CCP algorithm implementations relative to traditional in-kernel implementations.

4.3.3 Supporting per-ACK Logic Within the Datapath

How must the datapath provide congestion feedback to algorithms running in the CCP agent? Ideally, a datapath should supply congestion signals to algorithms with suitable granularity (e.g., averaged over an RTT, rather than per ACK), at configurable time intervals (e.g., a few times every RTT) and during critical events (e.g., packet losses). With CCP, users can specify such datapath behavior using a domain-specific language (§4.4). At a high level, CCP-compatible datapaths expose a number of *congestion signals*, over which users can write *fold functions* to summarize network observations for algorithms. It is also possible to perform *control actions* such as reporting summarized measurements to CCP or setting a flow's pacing rate. Datapath programs can trigger fold functions and control actions when certain conditions hold, e.g., an ACK is received or a timer elapses. Users can thus control how to partition the logic of the algorithm between these two components according to their performance and flexibility requirements (§4.4.4).

4.4 Writing Algorithms in CCP

Figure 4.1 shows the control loop of a congestion control algorithm in CCP. Users implement two callback handlers (`onCreate()` and `onReport()`) in the CCP agent and one or more datapath programs. When a new flow is created, CCP's datapath component invokes the `onCreate()` handler. The implementation of `onCreate()` must install an initial datapath program for that flow. Datapath programs could compute summaries over per-packet congestion signals (such as a minimum packet delay or a moving average of packet delivery rate) and report summaries or high priority conditions

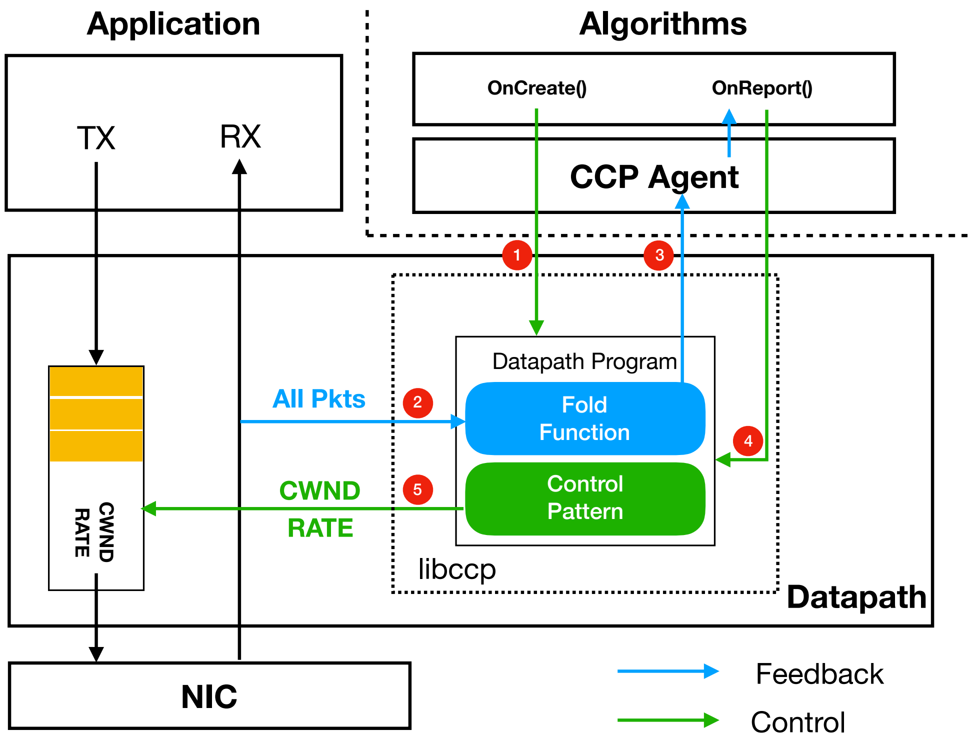


Figure 4.1: Congestion control algorithms in CCP are distinct from the application and datapath. Users specify an `onCreate()` handler which CCP calls when a new flow begins. In this handler, algorithms install (1) a datapath program. This datapath program aggregates incoming measurements (2) using user-defined fold functions and occasionally sends reports (3) to CCP, which calls the `onReport()` handler. The `onReport()` handler can update (4) the datapath program, which uses its defined control patterns to enforce (5) a congestion window or pacing rate.

```
(def (Report (volatile acked 0) (volatile lost 0)))
(when true
  (:= Report.acked (+ Report.acked Ack.bytes_acked))
  (:= Report.lost  (+ Report.lost  Ack.lost_pkts_sample))
  (fallthrough))
(when (> Report.lost 0) (report))
```

Listing 4.1: A simple datapath program to count bytes acked and report on losses.

(such as loss) to the CCP agent. On a report, the CCP agent invokes the `onReport()` handler which contains the bulk of the logic of the congestion control algorithm. The `onReport()` function computes and installs the flow's congestion window or sending rate using the signals from the datapath report. It may also replace the datapath program entirely with different logic.

4.4.1 Datapath Program Abstractions

CCP's datapath programs are written in a simple domain specific language. These programs exist in order to provide a per ACK execution environment, where algorithms can define and update variables per ACK and perform *control actions*, in response to the values of these variables.

Listing 4.1 shows a program that counts the cumulative number of packets acknowledged and lost and reports these counters immediately upon a loss. The first statement of the program allows users to define custom variables. The "Report" block signifies that these variables should be included in the report message sent to the CCP agent. The `volatile` marker means that these variables should be reset to their initial values, 0, after every report to the CCP agent.

Following the `def` block, *fold functions* provide custom summaries over primitive congestion signals. Datapath programs have read access to these *primitive congestion signals* (prefixed with "`Ack.`" or "`Flow.`" to specify their measurement period), which are exposed by the datapath on every incoming packet. Such signals include the round trip delay sample, the number of bytes the datapath believes have been dropped by the network, and the delivery rates of packets. Table 4.2 enumerates the primitive congestion signals we support. Users can write simple mathematical summaries over these primitive signals, as shown in Lines 3-4 of Listing 4.1.

Finally, algorithms can perform control actions in response to conditions defined by the fold function variables, e.g., updating a rate or cwnd or reporting the user defined variables to the CCP agent. As shown in Listing 4.1, the program defines a series of `when` clauses, and performs the following block only if the condition was evaluated to true.

CCP's datapath program language provides an event driven programming model. The condition `(when true...)` signifies that the body should be evaluated on every packet. This is where the program might calculate fold function summaries. The `when` clauses have access to all the fold function variables, as well as timing related counters. The `report` instruction causes the datapath to transmit the `acked` and `lost` counters to the CCP agent. By default, the program evaluates until one `when` clause evaluates to true; the `(fallthrough)` instruction at the end of the first `when` indicates that subsequent `when` clauses should also be evaluated.

4.4.2 CCP Algorithm Logic

The `onReport()` handler provides a way to implement congestion control actions in user-space in reaction to reports from the datapath. For example,

Primitive congestion signals	
Signal	**Definition**
`Ack.bytes_acked`, `Ack.packets_acked`	In-order acknowledged
`Ack.bytes_misordered`, `Ack.packets_misordered`	Out-of-order acknowledged
`Ack.ecn_bytes`, `Ack.ecn_packets`	ECN-marked
`Ack.lost_pkts_sample`	Number of lost packets
`Ack.now`	Datapath time (e.g., Linux `jiffies`)
`Flow.was_timeout`	Did a timeout occur?
`Flow.rtt_sample_us`	A recent sample RTT
`Flow.rate_outgoing`	Outgoing sending rate
`Flow.rate_incoming`	Receiver-side receiving rate
`Flow.bytes_in_flight`, `Flow.packets_in_flight`	Sent but not yet acknowledged

Operators	
Class	**Operations**
Arithmetic	`+`, `-`, `*`, `/`
Assignment	`:=`
Comparison	`==`, `<`, `>`, `or`, `and`
Conditionals	`If` (branching)

Variable Scopes	
Scope	**Description**
`Ack`	Signals measured per packet
`Flow`	Signals measured per connection
`Timer`	Multi-resolution timer that can be zeroed by a call to `reset`

Table 4.2: Datapath language: congestion signals, operators, and scopes.

a simple additive-increase multiplicative-decrease (AIMD) algorithm could be implemented in Python[1] using the `acked` and `lost` bytes reported every round-trip time from the datapath:

```
def onReport(self, report):
  if report["lost"] > 0:
     self.cwnd = self.cwnd / 2
  else:
     acked = report["acked"]
     self.cwnd = self.cwnd + acked*MSS/self.cwnd
  self.update("cwnd", self.cwnd/MSS)
```

We have implemented complex functionality within congestion control algorithms by leveraging slow-path logic, for example, a congestion control algorithm that uses Fast Fourier Transform (FFT) operations [51].

If the round-trip time of the network is a few milliseconds or more, it is possible to locate congestion control algorithm logic entirely within CCP with high fidelity relative to a per-packet update algorithm, as we show in §4.6.1.

4.4.3 Example: BBR

As a more involved example, we show below how various components of TCP BBR [20] are implemented using the CCP API. A BBR sender estimates the rate of packets delivered to the receiver, and sets its sending rate to the maximum delivered rate (over a sliding time window), which is believed to be the rate of the bottleneck link between the sender and the receiver.

This filter over the received rate is expressed simply in a fold function:

```
(when true
    (:= minrtt (min minrtt Ack.rtt_sample_us))
```

[1]Our CCP implementation is in Rust and exposes Python bindings (§4.5).

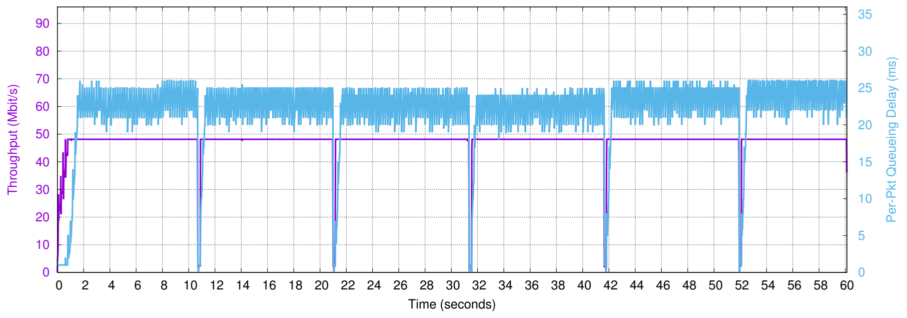


Figure 4.2: Our CCP implementation of BBR used for a bulk transfer over a 48 Mbit/s link with a 20 ms RTT and 2 BDPs of buffering. The bandwidth probe phase can be seen in the oscillation of the queueing delay, and the RTT probe phase can be seen in the periodic dips in throughput.

```
    (:= curr_btl_est (max curr_btl_est Flow.rate_incoming))
    (fallthrough))
```

To determine whether a connection can send more than its current sending rate, BBR probes for additional available bandwidth by temporarily increasing its sending rate by a factor (1.25×) of its current sending rate. To drain a queue that may have been created in the process, it also reduces its rate by a reciprocal factor (0.75×) before starting to send at the new estimated bottleneck link rate.

The following excerpt expresses this sending pattern (for simplicity, we show only 2 transitions):

```
(when (== pulseState 0)
  (:= Rate (* 1.25 curr_btl_est))
  (:= pulseState 1))
(when (&& (== pulseState 1)
          (> Timer.micros Flow.rtt_sample_us))
  (:= Rate (* 0.75 curr_btl_est))
  (:= pulseState 2))
```

Here, the variable `pulseState` denotes the state of the sender's bandwidth probing: probing with high sending rate (0) and draining queues with low sending rate (1). Each `when` clause represents a pulse state transition and is conditioned on the resettable timer `Timer.micros`. Upon the transition, the handler sets the `Rate` and advances `pulseState`. After the last phase of the pulse, the handler would reset the timer and `pulseState` to restart the sending pattern (not shown).

Figure 4.2 shows the impact of BBR's bandwidth probing[2] on the achieved goodput and queueing delays when a single flow runs over a 48 Mbit/s bottleneck link with a 20 ms round trip propagation delay. BBR's windowed min/max operations and the RTT probing phase (showing steep rate dips every 10 seconds) are implemented in the slow path's `onReport()` handler by installing a new fold function. CCP's split programming model enables this flexible partitioning of functionality.

4.4.4 Case Study: Slow Start

Because algorithms no longer make decisions upon every ACK, CCP changes the way in which developers should think about congestion control, and correspondingly provides multiple implementation choices. As a result, new issues arise about where to place algorithm functionality. We discuss the involved trade-offs with an illustrative example: slow start.

Slow start is a widely used congestion control module in which a connection probes for bandwidth by multiplicatively increasing its congestion window (`cwnd`) every RTT. Most implementations increment `cwnd` per ACK,

[2]We only implement BBR's PROBE_BW and PROBE_RTT. Our implementation is here: `github.com/ccp-project/bbr`.

```
fn create(...) {
  datapath.install("
  (def (Report (volatile acked 0) (volatile loss 0)))
  (when true
    (:= Report.acked (+ Report.acked Ack.bytes_acked)))
  (when (> Micros Flow.rtt_sample_us) (report) (reset))");
}
fn onReport(...) {
  if report.get_field("Report.loss") == 0 {
    let acked = report.get_field("Report.acked");
    self.cwnd += acked;
    datapath.update_field(&[("Cwnd", self.cwnd)]);
  } else { /* exit slow start */ }
}
```

Listing 4.2: A CCP implementation of slow start.

either by the number of bytes acknowledged in the ACK, or by 1 MSS. One way to implement slow start is to retain the logic entirely in CCP, and measure the size of the required window update from datapath reports. We show an example in Listing 4.2. This implementation strategy is semantically closest in behavior to native datapath implementations.

For some workloads this approach may prove problematic, depending on the parameters of the algorithm. If the reporting period defined is large, then infrequent slow start updates can cause connections to lose throughput. Figure 4.3 demonstrates that, on a 48 Mbps, 100 ms RTT link, different implementations of slow start exhibit differing window updates relative to the Linux kernel baseline. A version with a 1-RTT reporting period lags behind the native datapath implementation. It is also possible to implement slow start within the datapath either by using congestion window increase (Listing 4.3), or by using rate based control:

```
(when (> Timer.Micros Flow.rtt_sample_us)
```

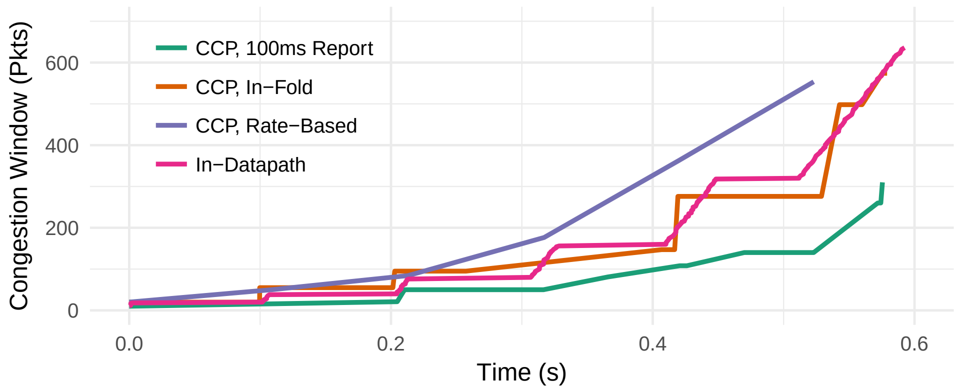


Figure 4.3: Different implementations of slow start have different window update characteristics. The control pattern implementation is rate-based, so we show the congestion window corresponding to the achieved throughput over each RTT.

```
(:= Rate (* Rate 2))
(:= Timer.Micros 0))
```

Take-away. As outlined in §4.3, the programming model of datapath programs is deliberately limited. First, we envision that in the future, CCP will support low-level hardware datapaths—the simpler the fold function execution environment is, the easier these hardware implementations will be. Second, algorithms able to make complex decisions on longer time-scales will naturally do so to preserve cycles for the application and datapath; as a result, complex logic inside the fold function may not be desirable.

More broadly, developers may choose among various points in the algorithm design space. On one extreme, algorithms may be implemented almost entirely in CCP, using the fold function as a simple measurement query language. On the other extreme, CCP algorithms may merely specify transitions between in-datapath fold functions implementing the primary

```
fn create(...) {
  datapath.install("
  (def (volatile Report.loss 0))
  (when true (:= Cwnd (+ Cwnd Ack.bytes_acked)))
  (when (> Ack.lost_pkts_sample 0) (report))");
}
fn onReport(...) { /* exit slow start */ }
```

Listing 4.3: A within-fold implementation of slow start. Note that CCP algorithm code is not invoked at all until the connection experiences its first loss.

control logic of the algorithm. Ultimately, users are able to choose the algorithm implementation best suited to their congestion control logic and application needs.

4.5 CCP Implementation

We implement a user-space CCP agent in Rust, called Portus[3], which implements functionality common across independent congestion control algorithm implementations, including a compiler for the datapath language and a serialization library for IPC communication. CCP congestion control algorithms are hence implemented in Rust; we additionally expose bindings in Python. The remainder of this section will discuss datapath support for CCP.

4.5.1 Datapath Requirements

A CCP-compatible datapath must accurately enforce the congestion control algorithm specified by the user-space CCP module. Once a datapath imple-

[3]github.com/ccp-project/portus

ments support for CCP, it automatically enables all CCP algorithms. An implementation of the CCP datapath must perform the following functions:

- The datapath should communicate with a user-space CCP agent using an IPC mechanism. The datapath multiplexes `report`s from multiple connections onto the single persistent IPC connection to the slow path. It must also perform the proper serialization for all messages received and sent.
- The datapath should execute the user-provided domain-specific program on the arrival of every acknowledgment or a timeout in a safe manner. Datapath programs (§4.4) may include simple computations to summarize per-packet congestion signals (Table 4.2) and enforce congestion windows and rates.

4.5.2 Safe Execution of Datapath Programs

Datapaths are responsible for safely executing the program sent from the user-space CCP module. While CCP will compile the instructions and check for mundane errors (e.g., use of undefined variables) before installation, it is the datapath's responsibility to ensure safe interpretation of the instructions. For example, datapaths should prevent divide by zero errors when calculating user defined variables and guarantee that programs cannot overwrite the congestion primitives. However, algorithms are allowed to set arbitrary congestion windows or rates, in the same way that any application can congest the network using UDP sockets.

Thankfully, this task is straightforward as datapath programs are limited in functionality: programs may not enter loops, perform floating point operations, define functions or data structures, allocate memory, or use pointers.

Rather, programs are strictly a way to express arithmetic computations over a limited set of primitives, define when and how to set congestion windows and pacing rates, and report measurements.

4.5.3 `libccp`: CCP's Datapath Component

We have implemented a library, `libccp`[4], that provides a reference implementation of CCP's datapath component, in order to simplify CCP datapath development. `libccp` is lightweight execution loop for datapath programs and message serialization. While we considered using eBPF [35] or TCP BPF [13] as the execution loop, including our own makes `libccp` portable to datapaths outside the Linux kernel; the execution loop runs the same code in all three datapaths we implemented.

To use `libccp`, the datapath must provide callbacks to functions that: (1) set the window and rate, (2) provide a notion of time, and (3) send an IPC message to CCP. Upon reading a message from CCP, the datapath calls `ccp_recv_msg()`, which automatically de-multiplexes the message for the correct flow. After updating congestion signals, the datapath can call `ccp_invoke()` to run the datapath program, which may update variable calculations, set windows or rates, and send report summaries to CCP. It is the responsibility of the datapath to ensure that it correctly computes and provides the congestion signals in Table 4.2.

The more signals a datapath can measure, the more algorithms that datapath can support. For example, CCP can only support DCTCP [2] or ABC [50] on datapaths that provide ECN support; CCP will not run algorithms on datapaths lacking support for that algorithm's requisite primitives.

[4]`github.com/ccp-project/libccp`

4.5.4 Datapath Implementation

We use `libccp` to implement CCP support in three software datapaths: the Linux kernel[5]; mTCP, a DPDK-based datapath; and Google's QUIC. For both the Linux kernel and QUIC datapaths, we leveraged their respective pluggable congestion control interfaces, which provide callbacks upon packet acknowledgements and timeouts, where the `libccp` program interpreter can be invoked. The kernel module implements the communication channel to CCP using either Netlink sockets or a custom character device, while mTCP and QUIC use Unix domain sockets. We additionally modified the QUIC source code to support multiplexing CCP flows on one persistent IPC connection and to expose the function callbacks required by the `libccp` API.

Unlike QUIC and the Linux kernel, mTCP only implements Reno and does not explicitly expose a congestion control interface for new algorithms. In order to achieve behavior consistent with other datapaths, we also implemented SACK and packet pacing; these features were previously lacking.

The definition of congestion signal primitives, IPC, and window and rate enforcement mechanisms is the only datapath-specific work needed to support CCP. As an example, Table 4.3 details the mapping of kernel variables to CCP primitives. Most of these definitions are straightforward; the CCP API merely requires datapaths to *expose* variables they are already measuring. All other necessary functionality, most notably interpreting and running the datapath programs, is shared amongst software datapaths via `libccp` (§4.5.3).

[5]Our kernel module is built on Linux 4.14: github.com/ccp-project/ccp-kernel

Signal	Definition
`Ack.bytes_acked`, `Ack.packets_acked`	`Delta(tcp_sock.bytes_acked)`
`Ack.bytes_misordered`, `Ack.packets_misordered`	`Delta(tcp_sock.sacked_out)`
`Ack.ecn_bytes`, `Ack.ecn_packets`	`in_ack_event: CA_ACK_ECE`
`Ack.lost_pkts_sample`	`rate_sample.losses`
`Ack.now`	`getnstimeofday()`
`Flow.was_timeout`	`set_state: TCP_CA_Loss`
`Flow.rtt_sample_us`	`rate_sample.rtt_us`
`Flow.rate_outgoing`	`rate_sample.delivered / Delta(tcp_sock.first_tx_mstamp)`
`Flow.rate_incoming`	`rate_sample.delivered / Delta(tcp_sock.tcp_mstamp)`
`Flow.bytes_in_flight`, `Flow.packets_in_flight`	`tcp_packets_in_flight(tcp_sock)`

Table 4.3: Definition of CCP primitives in terms of the `tcp_sock` and `rate_sample` structures, for the Linux kernel datapath.

4.6 Evaluation

We evaluated the following aspects of CCP:

Fidelity (§4.6.1). Do algorithms implemented in CCP behave similarly to algorithms implemented within the datapath? Using the Linux kernel datapath as a case study, we explore both achieved throughput and delay for persistently backlogged connections as well as achieved flow completion time for dynamic workloads.

Overhead of datapath communication (§4.6.2). How expensive is communication between CCP and the datapath?

High bandwidth, low RTT (§4.6.3). We use ns-2 simulations to demonstrate that CCP's method of taking congestion control actions periodically can perform well even in ultra-low RTT environments.

Unless otherwise specified, we evaluated our implementation of CCP using Linux 4.14.0 on a machine with four 2.8 Ghz cores and 64 GB memory.

4.6.1 Fidelity

The Linux kernel is the most mature datapath we consider. Therefore, we present an in-depth exploration of congestion control outcomes comparing CCP and native-kernel implementations of two widely used congestion control algorithms: NewReno [58] and Cubic [54]. As an illustrative example, Figure 4.4 shows one such comparison of congestion window update decisions over time on an emulated 96 Mbit/s fixed-rate Mahimahi [95] link with a 20 ms RTT. We expect and indeed observe minor deviations as the connection progresses and small timing differences between the two implementations cause the window to differ, but overall, not only does CCP's implementation of Cubic exhibit a window update consistent with a cubic increase function, but its updates closely match the kernel implementation.

For the remainder of this subsection, we compare the performance of CCP and kernel implementations of NewReno and Cubic on three metrics (throughput and delay in §4.6.1, and FCT in §4.6.1) and three scenarios, all using Mahimahi.

Throughput and Delay.

We study the following scenarios:

Fixed-rate link ("fixed"). A 20 ms RTT link with a fixed 96 Mbit/s rate and 1 BDP of buffering.

Cellular link ("cell"). A 20 ms RTT variable-rate link with a 100-packet buffer based on a Verizon LTE bandwidth trace [95].

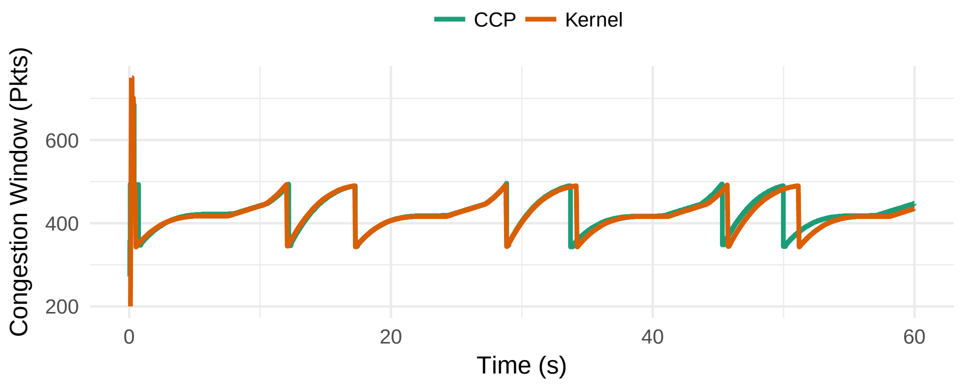


Figure 4.4: Cubic in CCP matches Cubic in Linux TCP.

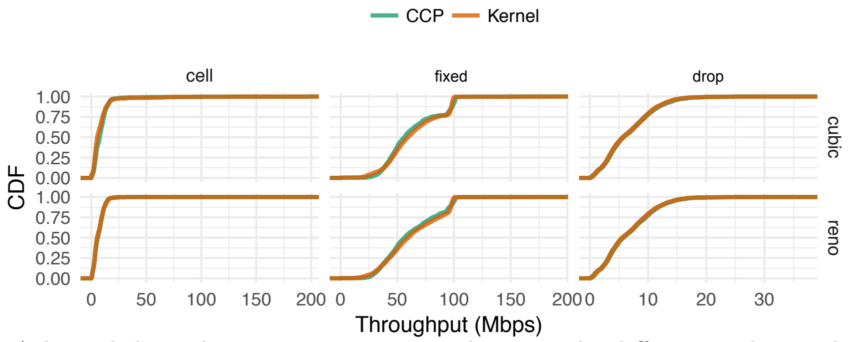
Stochastic drops ("drop"). A 20 ms RTT link with a fixed 96 Mbit/s rate, but with 0.01% stochastic loss and an unlimited buffer. To ensure that both tested algorithms encountered exactly the same conditions, we modified Mahimahi to use a fixed random seed when deciding whether to drop a packet.

These three scenarios represent a variety of environments congestion control algorithms encounter in practice, from predictable to mobile to bufferbloated paths. We calculate, per-RTT over twenty 1-minute experiments, the achieved throughput (4.5a) and delay (4.5b), and show the ensuing distributions in Figure 4.5.

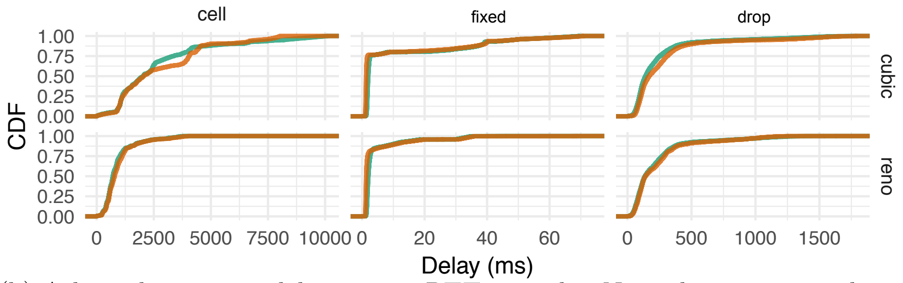
Overall, both distributions are close, suggesting that CCP's implementations make the same congestion control decisions as the kernel.

Flow Completion Time.

To measure flow completion times (FCT), we use a flow size distribution compiled from CAIDA Internet traces [17] in a similar setting to the "fixed"



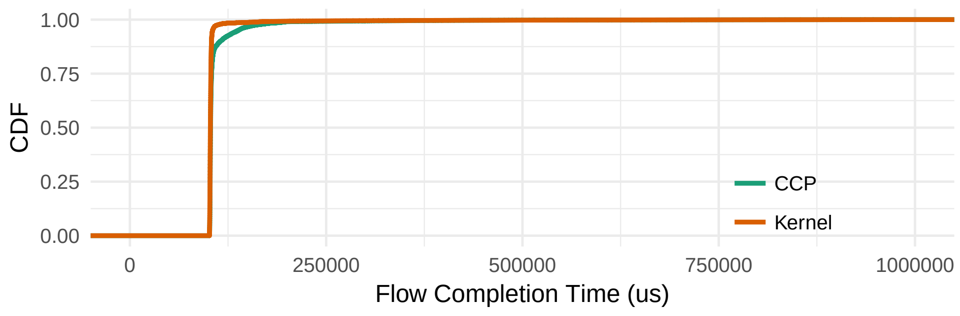
(a) Achieved throughput over 1 RTT periods. Note the different scales on the x-axes for the three scenarios.



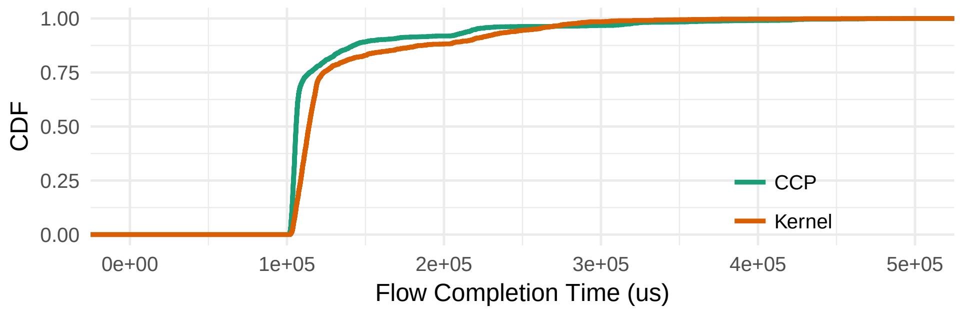
(b) Achieved queueing delay over 1 RTT periods. Note the varying scales on the x-axes for the three scenarios.

Figure 4.5: Comparison of achieved throughput over 20 ms periods. The achieved throughput distributions are nearly identical across the three scenarios and two congestion control algorithms evaluated.

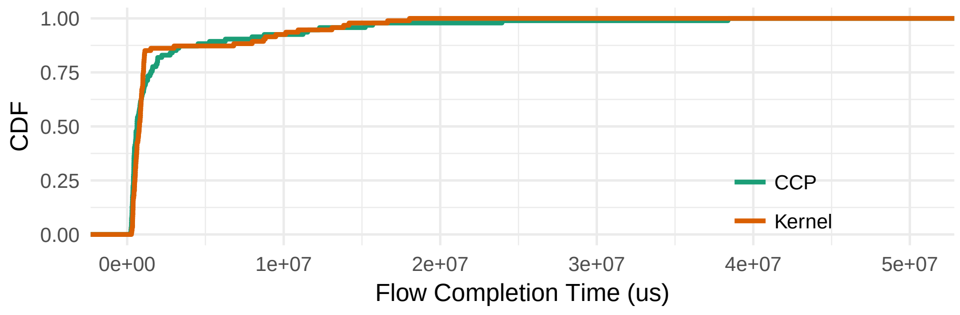
scenario above; we use a 100 ms RTT and a 192 Mbit/s link. To generate traffic, we use a traffic generator to sample flow sizes from the distribution and send flows of that size according to a Poisson arrival process to a single client behind the emulated Mahimahi link. We generate flows with 50% average link load, and generate $100,000$ flows to the client from 50 sending servers using persistent connections to the client. We used Reno as the congestion control algorithm in both cases. To ensure that the kernel-native congestion control ran under the same conditions as the CCP implementa-



(a) 0-10KB Flows



(b) 10KB-1MB Flows



(c) 1MB+ Flows

Figure 4.6: CDF comparisons of flow completion times. Note the differing x-axes.

tion, we disabled the slow-start-after-idle option.

Of the 100, 000 flows we sampled from the CAIDA workload, 97, 606 were 10 KB or less, comprising 487 MB, while the 95 flows greater than 1 MB in size accounted for 907 MB out of the workload's total of 1.7 GB.

Across all flow sizes, CCP achieves FCTs 0.02% lower than the kernel in the median, 3% higher in the 75th percentile, and 30% higher in the 95th percentile.

Small flows. Flows less than 10 KB in size, shown in Figure 4.6a, are essentially unaffected by congestion control. These flows, the vast majority of flows in the system, complete before either CCP algorithms or kernel-native algorithms make any significant decisions about them.

Medium flows. Flows between 10 KB and 1 MB in size, in Figure 4.6b achieve 7% lower FCT in the median with CCP because CCP slightly penalizes long flows due to its slightly longer update period, freeing up bandwidth for medium size flows to complete.

Large flows. CCP penalizes some flows larger than 1 MB in size compared to the native-kernel implementation: 22% worse in the median (Figure 4.6c).

4.6.2 Performance

Measurement Staleness.

Because our CCP implementation, Portus, runs in a different address space than datapath code, there is some delay between the datapath gathering a report and algorithm code acting upon the report. In the worst case, a severely delayed measurement could cause an algorithm to make an erroneous window update.

Fortunately, as Figure 4.7 shows, this overhead is small. We calculate

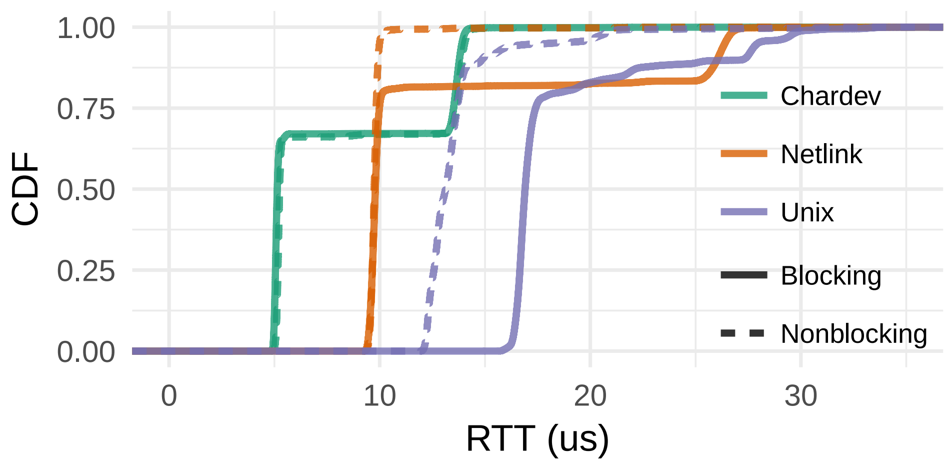
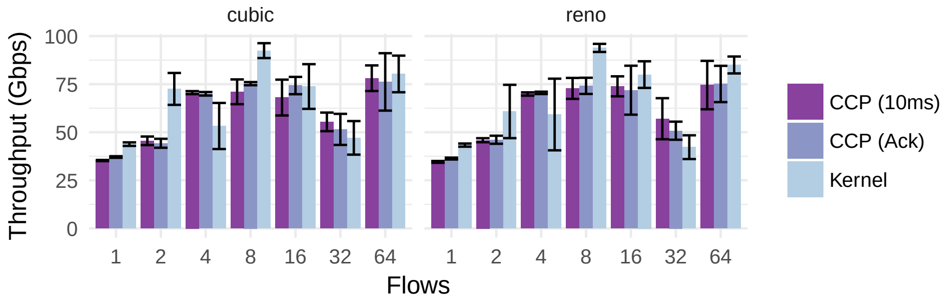


Figure 4.7: Minimum time required to send information to the datapath and receive a response using different IPC mechanisms.

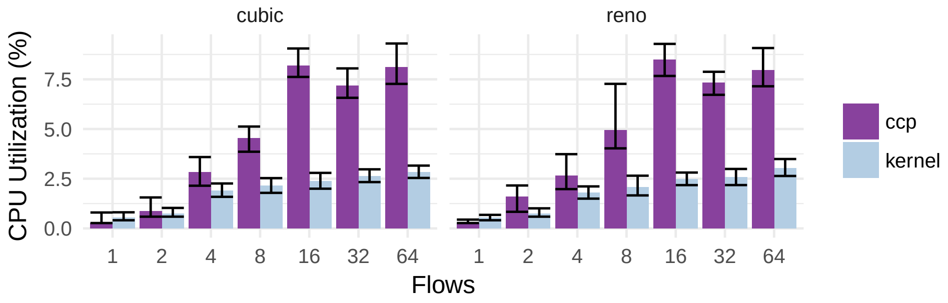
an IPC RTT by sending a time-stamped message to a kernel module (or user-space process in the case of a Unix-domain socket). The receiver then immediately echoes the message, and we measure the elapsed time at the originating process.

We test three IPC mechanisms: Unix-domain sockets [113], a convenient and popular IPC mechanism used for communication between user-space processes; Netlink sockets [117], a Linux-specific IPC socket used for communication between the kernel and user-space; and a custom kernel module, which implements a message queue that can be accessed (in both user-space and kernel-space) via a character device.

In all cases, the 95$^{\text{th}}$ percentile latency is less than 30 μs.



(a) Achieved localhost throughput as the number of flows increases



(b) CPU Utilization when saturating a 10 Gbit/s link.

Figure 4.8: CCP can handle many concurrent flows without significant CPU overhead. Error bars show standard deviation.

Scalability.

CCP naturally has nonzero overhead since more context switches must occur to make congestion control decisions in user-space. We test two scenarios as the number of flows in the system increases exponentially from 1 to 64. In both scenarios, we test CCP's implementation of Reno and Cubic against the Linux kernel's. We measure average throughput and CPU utilization in 1 second intervals over the course of 10 30-second experiments using iperf [128]. We evaluate CCP with two fold functions: one which implements a reporting interval of 10 ms, and another which reports on every packet.

We omit mTCP and QUIC from these scalability micro-benchmarks and focus on the kernel datapath. The QUIC toy server is mainly used for integration testing and does not perform well as the number of flows increase; we confirmed this behavior with Google's QUIC team. Similarly, after discussion with the mTCP authors, we were unable to run mTCP at sufficient speeds to saturate a localhost or 10 Gbit/sec connection.

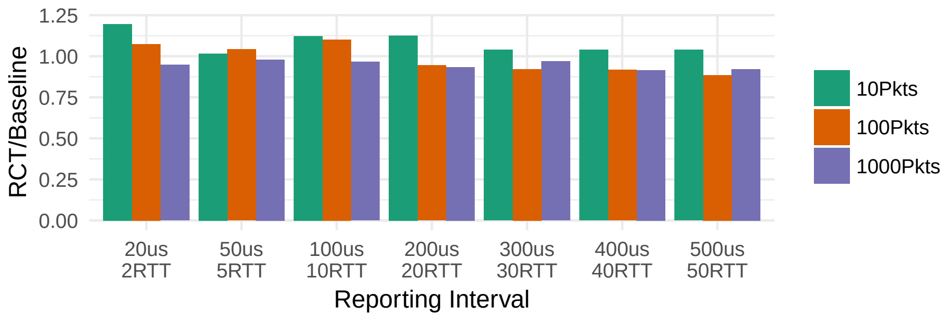
Localhost microbenchmark. We measure achieved throughput on a loopback interface as the number of flows increases. As the CPU becomes fully utilized, the achieved throughput will plateau. Indeed, in Figure 4.8a, CCP matches the kernel's throughput up to the maximum number of flows tested, 64.

CPU Utilization. To demonstrate the overhead of CCP in a realistic scenario, we scale the number of flows over a single 10 Gbit/s link between two physical servers and measure the resulting CPU utilization. Figure 4.8b shows that as the number of flows increases, the CPU utilization in the CCP case rises steadily. The difference between CCP and the kernel is most pronounced in the region between 16 and 64 flows, where CCP uses 2.0× as much CPU than the kernel on average; the CPU utilization nevertheless remains under 8% in all cases.

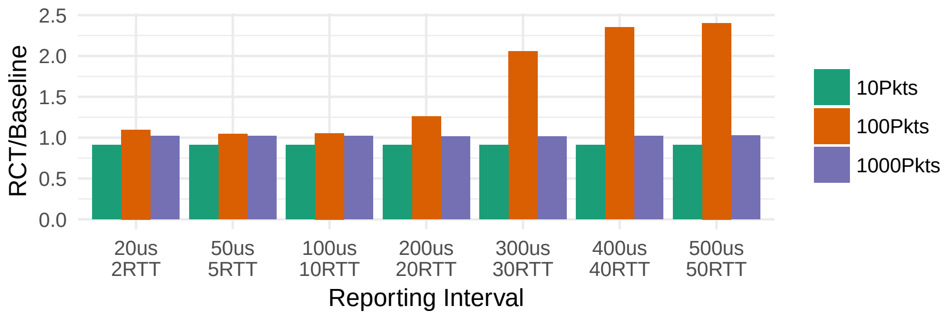
In both the CPU utilization and the throughput micro-benchmarks, the differences in CPU utilization stem from the necessarily greater number of context switches as more flows send measurements to CCP. Furthermore, the congestion control algorithm used does not affect performance.

4.6.3 Low-RTT and High Bandwidth Paths

To demonstrate it is feasible to separate congestion control from the datapath even in low-RTT and high bandwidth situations, we simulate a dat-



(a) Tail flow completion time at 10 Gbit/s



(b) Tail flow completion time at 40 Gbit/s

Figure 4.9: Mean tail completion across 50 simulations. While at 10 Gbit/s even rare reporting (every 50 RTTs) has limited overhead (at most 20%), at 40 Gbit/s, a 1 ms reporting period is necessary to avoid performance degradation.

acenter incast scenario using ns-2 [99]. We model CCP by imposing both forms of delays due to CCP: (i) the period with which actions can be taken (the reporting period) and, (ii) the staleness after which sent messages arrive in CCP. We used our microbenchmarks in §4.6.2 to set the staleness to 20 μs, and vary the reporting interval since it is controlled by algorithm implementations. We used a 20 μs RTT with a 50-to-1 incast traffic pattern across 50 flows with link speeds of 10 and 40 Gbit/s. To increase the statistical significance of our results, we introduce a small random jitter to flow start times (<10μs with 10 Gbit/s bandwidth and <2.5 μs with 40 Gbit/s bandwidth) and run each point 50 times with a different simulator random seed value and report the mean.

Figure 4.9 compares the results with the baseline set to in-datapath window update. We find that at 10 Gbit/s, CCP performance stays within 15% of the baseline across different flow sizes and reporting intervals ranging from 10 μs to 500 μs. Recall that 500 μs is 50$\times$ the RTT; even this infrequent reporting period yields only minor degradation.

Meanwhile, at 40 Gbit/s the slowdown over the baseline increases with the reporting interval in the case of 100 packet flows, but not with 10 or 1000 packet flows. Similar to the results in §4.6.1, the short flows and long flows are both unaffected by the reporting period because the short flows complete too quickly and the long flows spend much of their time with large congestion windows regardless of the window update. Indeed, at 100 μs (10 RTTs), the tail completion time is within 10% of the baseline; as the reporting increases, the tail completion time increases to over 2$\times$ the baseline. This nevertheless suggests that when reporting intervals are kept to small multiples of the RTT, tail completion time does not suffer.

4.7 New Capabilities

We present four new capabilities enabled by CCP: new congestion control algorithms that use sophisticated user-space programming libraries, rapid development and testing of algorithms, congestion control for flow aggregates, and the ability to write an algorithm once and run it on multiple datapaths.

4.7.1 Sophisticated Congestion Control Algorithms

CCP makes it possible to use sophisticated user-space libraries, such as libraries for signal processing, machine learning, etc. to implement congestion control algorithms.

One example is Nimbus [51], a new congestion control algorithm that detects whether the cross traffic at a bottleneck link is elastic (buffer-filling) or not, and uses different control rules depending on the outcome. The Nimbus algorithm involves sending traffic in an asymmetric sinusoidal pulse pattern and using the sending and receiving rates measured over an RTT to produce a time-series of cross-traffic rates. The method then computes the FFT of this time-series and infers elasticity if the FFT at particular frequencies is large.

The implementation of Nimbus uses CCP to configure the datapath to report the sending and receiving rates periodically (e.g., every 10 ms), maintains a time-series of the measurements in user-space, and performs FFT calculations using a FFT library in Rust [115].

Although it is possible to implement such algorithms directly in the datapath, it would be significantly more difficult. For instance, one would need to implement the FFT operations with fixed-point arithmetic. Moreover, implementing the algorithm outside the datapath using CCP allows for a tighter development-testing loop than writing kernel code.

We anticipate that in the future, CCP will enable the use of other similarly powerful but computationally-intensive methods such as neural networks.

4.7.2 Velocity of Development

Copa [7] is a recently proposed model-based congestion control algorithm that seeks to maintain a target rate that is inversely proportional to the queuing delay, estimated as the difference of the current RTT and the minimum RTT. It is robust to non-congestive loss, buffer-bloat, and unequal propagation delays. It includes mechanisms to provide TCP competitiveness, accurate minimum RTT estimation, and imperfect pacing.

The authors of Copa used CCP to implement Copa recently, and in the process discovered a small bug that produced an erroneous minimum RTT estimate due to ACK compression. They solved this problem with a small modification to the Copa datapath program, and in a few hours were able to improve the performance of their earlier user-space implementation. The improvement is summarized here:

Algorithm	Throughput	Mean queue delay
Copa (UDP)	1.3 Mbit/s	9 ms
Copa (CCP-Kernel)	8.2 Mbit/s	11 ms

After the ACK compression bug was fixed in the CCP version, Copa achieves higher throughput on a Mahimahi link with 25 ms RTT and 12 Mbit/s rate while maintaining low mean queueing delay. Because of ACK compression, the UDP version over-estimates the minimum RTT by 5×.

4.7.3 Flow Aggregation

Congestion control on the Internet is performed by individual TCP connections. Each connection independently probes for bandwidth, detects congestion on its path, and reacts to it. Congestion Manager [8] proposed the idea of performing congestion control for aggregates of flows at end-hosts. Flow aggregation allows different flows to share congestion information and achieve the correct rate more quickly.

We describe how to use CCP to implement a host-level aggregate controller that maintains a single aggregate window or rate for a group of flows

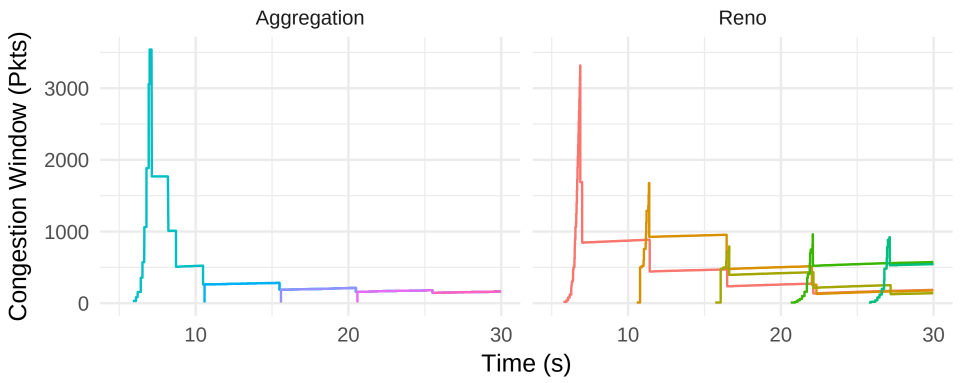


Figure 4.10: 5 20-second iperf flows with 10 second staggered starts. While Reno (right) must individually probe for bandwidth for each new connection, an aggregating congestion controller is able to immediately set the connection's congestion window to the fair share value.

and allocates that to individual flows—all with no changes to the non-CCP parts of the datapath.

Interface. In addition to the `create()` and `onReport()` event handlers, we introduce two new APIs for aggregate congestion controllers: `create_-subflow()` and `aggregateBy()`. CCP uses `aggregateBy()` to classify new connections into aggregates. Then, it calls either the existing `create()` handler in the case of a new aggregate, or the `create_subflow()` handler in the case of an already active one.

These handlers are natural extensions of the existing per-flow API; we implemented API support for aggregation in 80 lines of code in our Rust CCP implementation (§4.6). Algorithms can aggregate flows using the connection 5-tuple, passed as an argument to `aggregateBy()`.

As a proof of concept, we implement an algorithm which simply aggre-

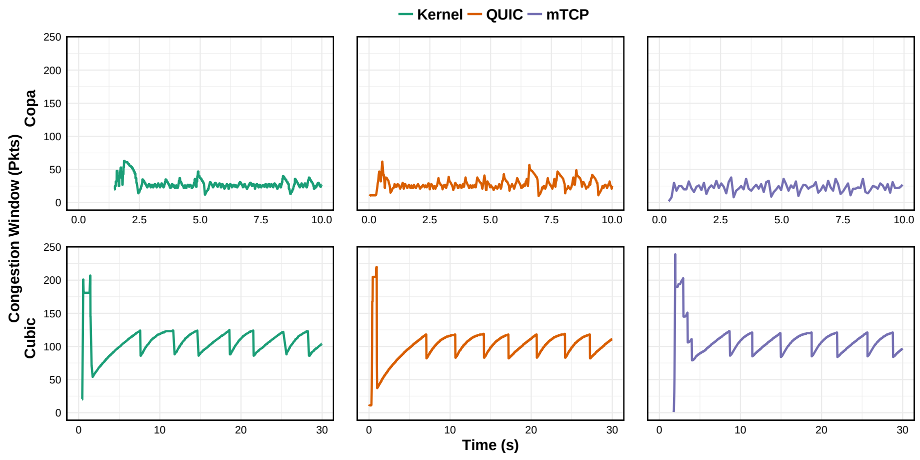


Figure 4.11: Comparison of the same CCP implementation of Cubic and Copa run on three different datapaths. Copa is run on a fixed 12 Mbps link with a 20 ms RTT; Cubic is run on a fixed 24 Mbps link with a 20 ms RTT.

gates all flows on each of the hosts's interfaces into one aggregate and assigns the window in equal portions to each sub-flow. Figure 4.10 shows the aggregator instantaneously apportioning equal windows to each flow in its domain.

4.7.4 Write-Once, Run-Anywhere

Implementing a new congestion control algorithm is difficult because of the subtle correctness and performance issues that require expertise to understand and resolve. New algorithms are often implemented in a single datapath and new datapaths have very few algorithms implemented. CCP enables algorithm designers to focus on building and testing a single solid implementation of their algorithm that users can then run on any (supported) datapath.

To exhibit this capability, we ran the same implementation of both Cubic (not previously implemented in mTCP) and Copa (§4.7.2, not previously implemented in any widely-used datapath[6]) on the three datapaths and plot the congestion window evolution over time in Figure 4.11.

As expected, the congestion window naturally evolves differently on each datapath, but the characteristic shapes of both algorithms are clearly visible. Copa uses triangular oscillations around an equilibrium of 1 BDP worth of packets (22 in this case), periodically draining the queue in an attempt to estimate the minimum RTT.

4.7.5 Park

We used CCP to implement congestion control support in Park [81], an open, extensible platform that presents a common RL interface to connect to a suite of computer system environments. These representative environments span a wide variety of problems across networking, databases, and distributed systems, and range from centralized planning problems to distributed fast reactive control tasks. In the backend, the environments are powered by both real systems (in 7 environments) and high fidelity simulators (in 5 environments). For each environment, Park defines the MDP formulation, e.g., events that triggers an MDP step, the state and action spaces and the reward function. This allows researchers to focus on the core algorithmic and learning challenges, without having to deal with low-level system implementation issues. At the same time, Park makes it easy to compare different proposed learning agents on a common benchmark, simi-

[6]In the time since we implemented Copa on CCP, a Copa implementation on mvfst, a QUIC datapath, has become available.

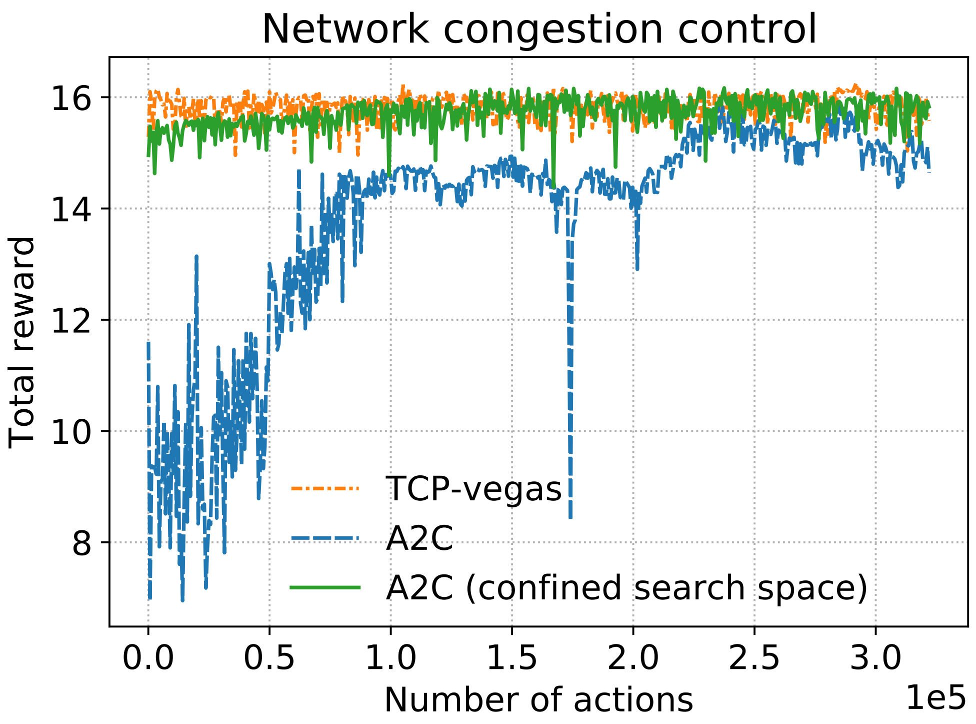


Figure 4.12: We used CCP to study the applicability of reinforcement learning to congestion control.

lar to how OpenAI Gym [101] has standardized RL benchmarks for robotics control tasks. Finally, Park defines a RPC interface between the RL agent and the backend system, making it easy to extend to more environments in the future.

We used CCP to implement Park's congestion control environment. We train and test the A2C agent in the centralized control setting (a single TCP connection) on a simple single-hop topology. We used a 48Mbps fixed-bandwidth bottleneck link with 50ms round-trip latency and a drop-tail buffer of 400 packets (2 bandwidth-delay products of maximum size packets) in each direction. For comparison, we run TCP Vegas [14]. Vegas attempts to maintain a small number of packets (by default, around 3) in the bottleneck queue, which results in an optimal outcome (minimal delay and packet loss, maximal throughput) for a single-hop topology without any competing traffic. “Confined search space” means we confine the action space of A2C agent to be only within 0.2 and 2× of the average action output from Vegas.

We find that in this environment, random exploration is inefficient to search the large state space that provides little reward gradient. This is because unstable control policies (which widely spans the policy space) cannot drain the network queue fast enough and results in indistinguishable (e.g., delay matches max queuing delay) poor rewards. Confining the search space with domain knowledge significantly improves learning efficiency in Figure 4.12.

4.7.6 Bundler

We demonstrate CCP’s flexibility by applying it as a core component of Bundler. Bundler [19] introduces the idea of *site-to-site* Internet traffic control. By “site”, we mean a single physical location with tens to many thousands of endpoints sharing access links to the rest of the Internet. Examples of sites include a company office, a coworking office building, a university campus, a single datacenter, and a point-of-presence (PoP) of a regional Internet Service Provider (ISP).

Consider a company site with employees running thousands of concurrent applications. The administrator may wish to enforce certain traffic control policies for the company; for example, ensuring rates and priorities for Zoom sessions, de-prioritizing bulk backup traffic, prioritizing interactive web sessions, and so on. There are two issues that stand in the way: first, the bottleneck for these traffic flows may not be in the company's network, and second, the applications could all be transiting different bottlenecks. So what is the company to do?

Cloud computing has made the second issue manageable. Because the cloud has become the prevalent method to deploy applications today, applications from different vendors often run from a small number of cloud sites (e.g., Amazon, Azure, etc.). This means that the network path used by these multiple applications serving the company's users are likely to share a common bottleneck; for example, all the applications running from Amazon's US-West datacenter, all the video sessions from a given Zoom datacenter, and so on. In this setting, by treating the traffic between the datacenter site and the company site as a single aggregate, the company's network administrator may be able to achieve their traffic control objectives.

But what about the first issue? The bottleneck for all the traffic between Amazon US-West and the company may not be the site's access link or at Amazon, but elsewhere, e.g., within the company's ISP; indeed, that may be the common case [26, 29, 77, 112]. Unfortunately, the company cannot control traffic when the queues build inside its ISP. And the ISP can't help because it does not know what the company's objectives are.[7]

Bundler solves this problem by enabling flexible control of a traffic *bundle* between a source site and a destination site by *shifting* the queues that would otherwise have accumulated elsewhere to the source's site (Figure 4.14). It then schedules packets from this shifted queue using standard techniques [24, 28, 39, 40, 87, 97, 105, 116, 121, 124, 142] to reduce mean flow-completion times, ensure low packet delays, isolate classes of traffic from

[7]Interdomain QoS mechanisms [12,138] have not succeeded in the Internet despite years of effort.

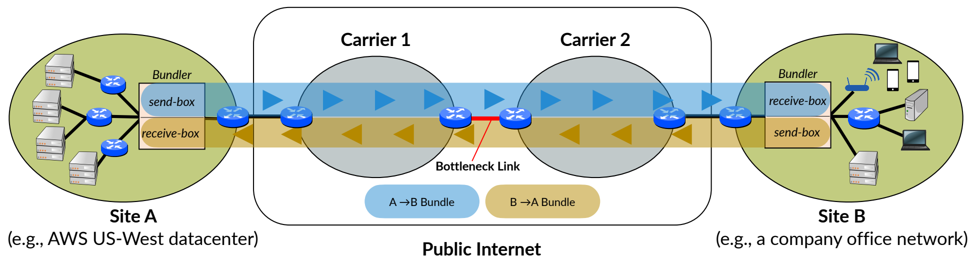


Figure 4.13: An example deployment scenario for Bundler in sites A and B. Traffic between the two boxes is aggregated into a single bundle, shown as shaded boxes. The sendbox schedules the traffic within the bundle according to the policy the administrator specifies (§4.7.6).

each other, etc.

The key idea in Bundler is a control loop between the source and destination sites to calculate the dynamic rate for the bundle. Rather than terminate end-to-end connections at the sites, we leave them intact and develop an "inner loop" control method between the two sites that computes this rate. The inner control loop uses a delay-based congestion control algorithm that ensures high throughput, but controls *self-inflicted queueing delays* at the actual bottleneck. By avoiding queues at the bottleneck, the source site can prioritize latency-sensitive applications and allocate rates according to its objectives.

By not terminating the end-to-end connections at the sites, Bundler achieves a key benefit: if the bottleneck congestion is due to other traffic not from the bundle, end-to-end algorithms naturally find their fair-share. It also simplifies the implementation because Bundler does not have to proxy TCP, QUIC, and other end-to-end protocols.

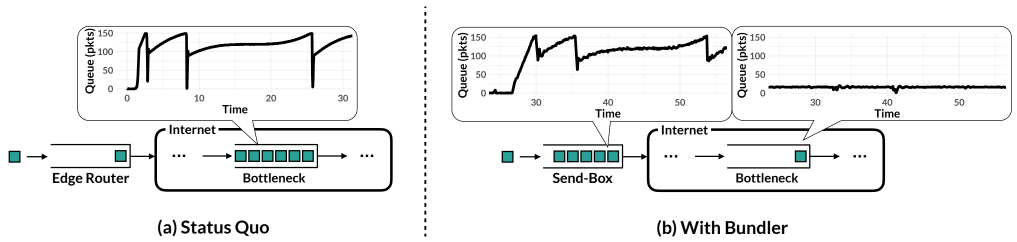


Figure 4.14: This illustrative example with a single flow shows how Bundler can take control of queues in the network. The plots, from measurements on an emulated path (as in §4.7.6), show the trend in queueing delays at each queue over time. The queue where delays build up is best for scheduling decisions, since it has the most choice between packets to send next. Therefore, the sendbox *shifts* the queues to itself.

Design

Recall that in order to do scheduling, we need to move the queues from the network to the Bundler. In this section, we first describe our key insight for moving the in-network queues, and then explain our specific design choices. Recall that each site deploys one Bundler middlebox which we logically partition into sender-side (sendbox) and receiver-side (receivebox) functionality.

We induce queuing at the sendbox by rate limiting the outgoing traffic. If this rate limit is made smaller than the bundle's fair share of bandwidth at the bottleneck link in the network, it will decrease throughput. Conversely, if the rate is too high, packets will pass through the sendbox without queueing. Instead, the rate needs to be set such that the bottleneck link sees a small queue while remaining fully utilized (and the bundled traffic competes fairly in the presence of cross traffic). We make a simple, but powerful, observation: existing congestion control algorithms calculate exactly this

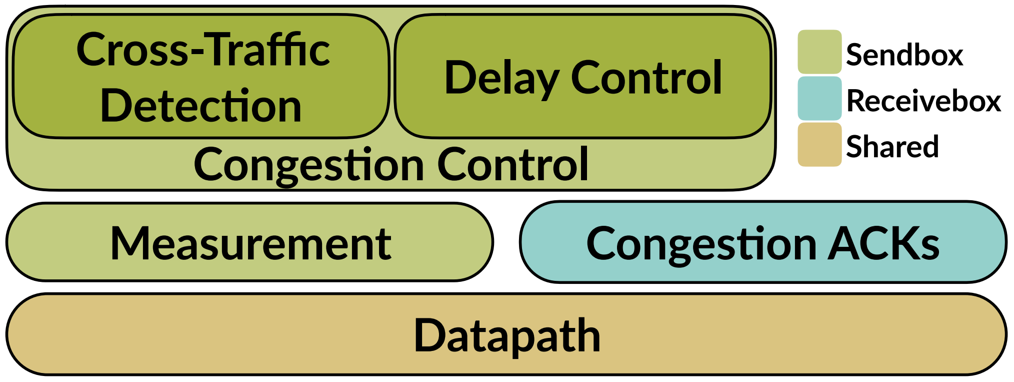


Figure 4.15: Bundler comprises of six sub-systems: four (in green) implement sendbox functionality, one (in blue) implements receivebox functionality, and the datapath (orange) is shared between the two.

rate [62]. Therefore, running such an algorithm to set a bundle's rate would reduce its self-inflicted queue at the bottleneck, causing packets to queue at the sendbox instead, without reducing the bundle's throughput. Note that end hosts would continue running a traditional congestion control algorithm as before (e.g., Cubic [54], BBR [20]) which is unaware of Bundler. Rather, the sendbox's congestion control algorithm acts on the traffic bundle as a *single unit*.

Figure 4.14 illustrates this concept for a single flow traversing a bottleneck link in the network. Without Bundler, packets from the end hosts are queued in the network, while the queue at the edge is unoccupied. In contrast, a Bundler deployed at the edge is able to shift the queue to its sendbox.

Figure 4.15 shows Bundler's sub-systems: (1) A congestion control module at the sendbox which implements the rate control logic and cross-traffic detection. (2) A mechanism for sending congestion feedback (ACKs) in the receivebox, and (3) a measurement module in the sendbox that computes congestion signals (RTT and receive rate) from the received feedback. We discuss options for implementing congestion feedback mechanism in §4.7.6 and how to use that feedback in the measurement module in §4.7.6. (4) A datapath for packet processing (which includes rate enforcement and packet scheduling). Any modern middlebox datapath, e.g., BESS [55], P4 [11], or Linux qdiscs, is suitable.

A congestion control algorithm at the sendbox, running atop CCP, would require network feedback from the receivers to measure congestion and adjust the sending rates accordingly. We discuss multiple options for obtaining this.

Passively observe in-band TCP acknowledgements. Conventional endhost-based implementations have used TCP acknowledgements to gather congestion control measurements. A simple strategy for Bundler is to passively observe the receiver generated TCP acknowledgements at the sendbox. However, we discard this option as it is specific to TCP and thus incompatible with alternate protocols, i.e., UDP for video streaming or QUIC's encrypted transport header [76].

Out-of-band feedback. Having eliminated the options for using in-band feedback, we adopt an out-of-band feedback mechanism: the receivebox sends out-of-band congestion ACKs to the sendbox. This decouples congestion signalling from traditional ACKs used for reliability and is thus indifferent to the underlying protocol (be it TCP, UDP, or QUIC).

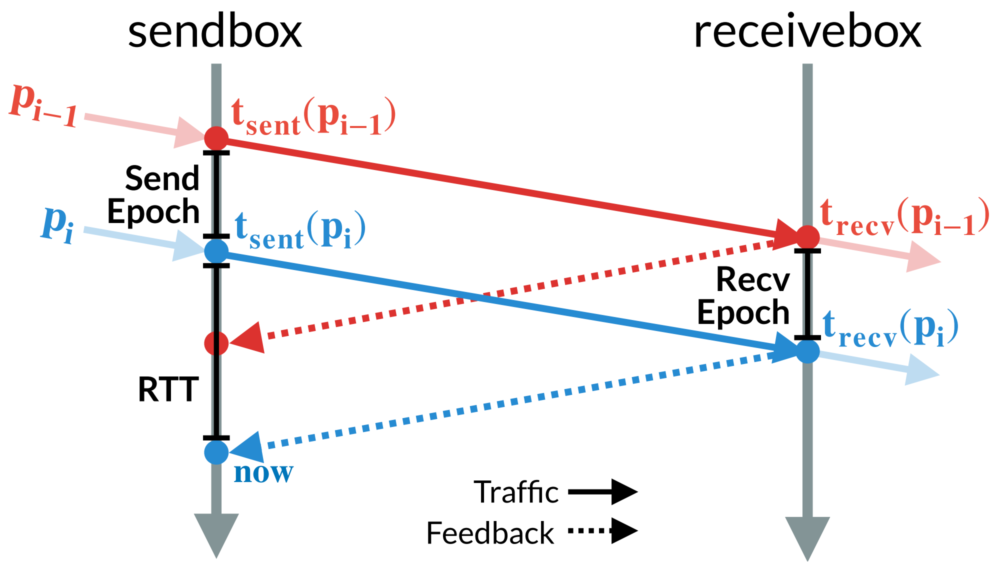


Figure 4.16: Example of epoch-based measurement calculation. Time moves from top to bottom. The sendbox records the packets that are identified as epoch boundaries. The receivebox, up on identifying such packets, sends a feedback message back to the sendbox, which allows it to calculate the RTT and epochs.

Measurement

Sending an out-of-band feedback message for every packet arriving at the receivebox would result in high communication overhead. Furthermore, conducting measurements on every outgoing packet at the sendbox would require maintaining state for each of them, which can be expensive, especially at high bandwidth-delay products. This overhead is unnecessary; reacting once per RTT is sufficient for congestion control algorithms [93]. The sendbox therefore samples a subset of the packets for which the receivebox sends congestion ACKs. We refer to the period between two successively sampled packets as an *epoch*, and each sampled packet as an *epoch boundary packet*.

The simplest way to sample an epoch boundary packet would be for the sendbox to probabilistically modify a packet (i.e., set a flag bit in the packet header) and the receivebox to match on this flag bit. However, where in the header should this flag bit be? Evolving packet headers has proved impractical [85], so perhaps we could use an encapsulation mechanism. Protocols at both L3 (e.g., NVGRE [45], IP-in-IP [106]) and L4 (e.g., VXLAN [80]) are broadly available and deployed in commodity routers today.

Happily, we observe that such packet modification is not inherently necessary; since the same packets pass through the sendbox and receivebox, uniquely identifying a given pattern of packets is sufficient to meet our requirements. In this scheme, the sendbox and receivebox both hash a subset of the header for every packet, and consider a packet as an epoch boundary if its hash is a multiple of the desired *sampling period*.

Upon identifying a packet p_i as an epoch boundary packet the sendbox records: (i) its hash, $h(p_i)$, (ii) the time when it is sent out, $t_{\text{sent}}(p_i)$, and (iii) the total number of bytes sent thus far including this packet, $b_{sent}(p_i)$. When the receivebox sees p_i, it also identifies it as an epoch boundary and sends a congestion ACK back to the sendbox. The congestion ACK contains $h(p_i)$ and the running count of the total number of bytes received for that bundle. Upon receiving the congestion ACK for p_i, the sendbox records the received information, and using its previously recorded state, computes the RTT and the rates at which packets are sent and received, as in Figure 4.16.

Epoch boundary identification. The packet header subset that is used for identifying epoch boundaries must have the following properties: (i) It must be the same at both the sendbox and the receivebox. (ii) Its values must remain unchanged as a packet traverses the network from the send-

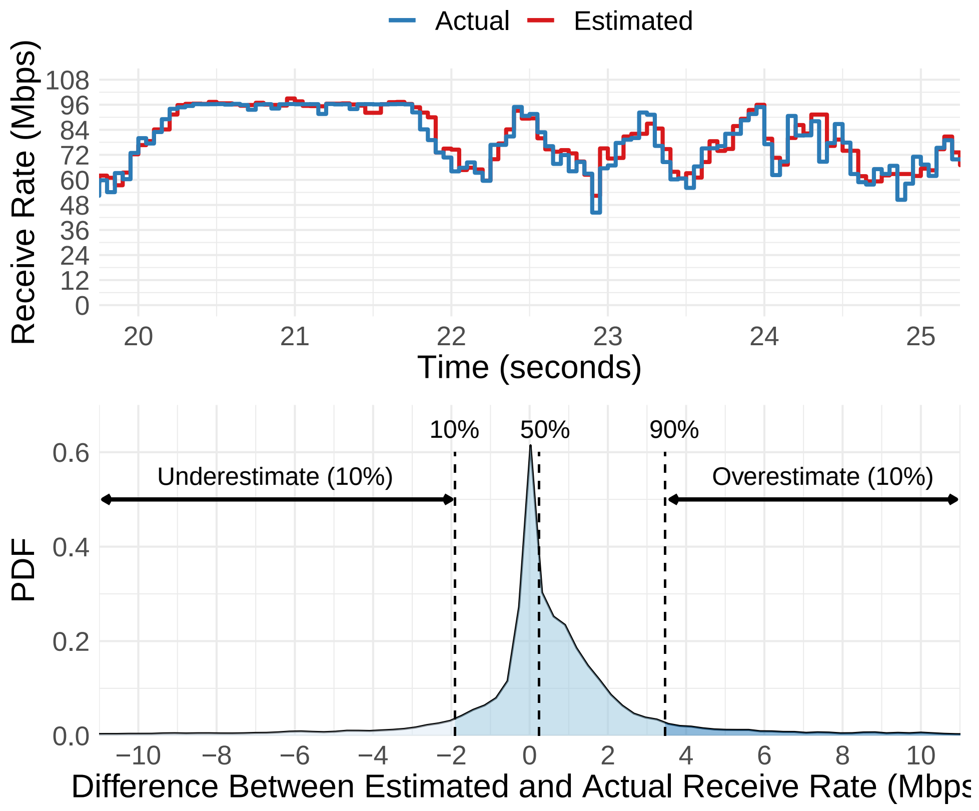


Figure 4.17: Bertha's estimate of the receive rate.

box to the receivebox (so, for example, the TTL field must be excluded).[8] (iii) It differentiates individual *packets* (and not just flows), to allow sufficient entropy in the computed hash values. (iv) It also differentiates a retransmitted packet from the original one, to prevent spurious samples from disrupting the measurements (this precludes, for example, the use of

[8]Certain fields, that are otherwise unchanged within the network, can be changed by NATs deployed within a site. Ensuring that the Bundler boxes sit outside the NAT would allow them to make use of those fields.

TCP sequence number). We expect that the precise set of fields used will depend on specific deployment considerations. For example, in our prototype implementation we use a header subset of the IPv4 IP ID field and destination IP and port. We make this choice for simplicity; it does not require tunnelling mechanisms and is thus easily deployable, and if Bundler fails, connections are unaffected. We note that previous proposals [118] have used IP ID for unique packet identification. The drawback of this approach is that it cannot be extended to IPv6. To support a wider set of scenarios, Bundler could use dedicated fields in an encapsulating header (as in [84]).

To visualize how these measurements impact the behavior of the signals over time we pick an experiment for which the median difference matches that of the entire distribution and plot a five second segment of our estimates compared to the actual values in Figure 4.17.

Choosing the epoch size. In order to balance reaction speed and overhead, epoch packets should be spaced such that measurements are collected approximately once per RTT [93]. Therefore, for each bundle, we track the minimum observed RTT ($minRTT$) at the sendbox and set the epoch size $N = (0.25 \times minRTT \times send\_rate)$, where the $send\_rate$ is computed as described above. The measurements passed to the congestion control algorithms at the sendbox are then computed over a sliding window of epochs that corresponds to one RTT. Averaging over a window of multiple epochs also increases resilience to possible re-ordering of packets between the sendbox and the receivebox, which can result in them seeing different number of packets between two epochs.

When the sendbox updates the epoch size N for a bundle, it needs to send an out-of-band message to the receivebox communicating the new value. To keep our measurement technique resilient to potential delay and loss of this

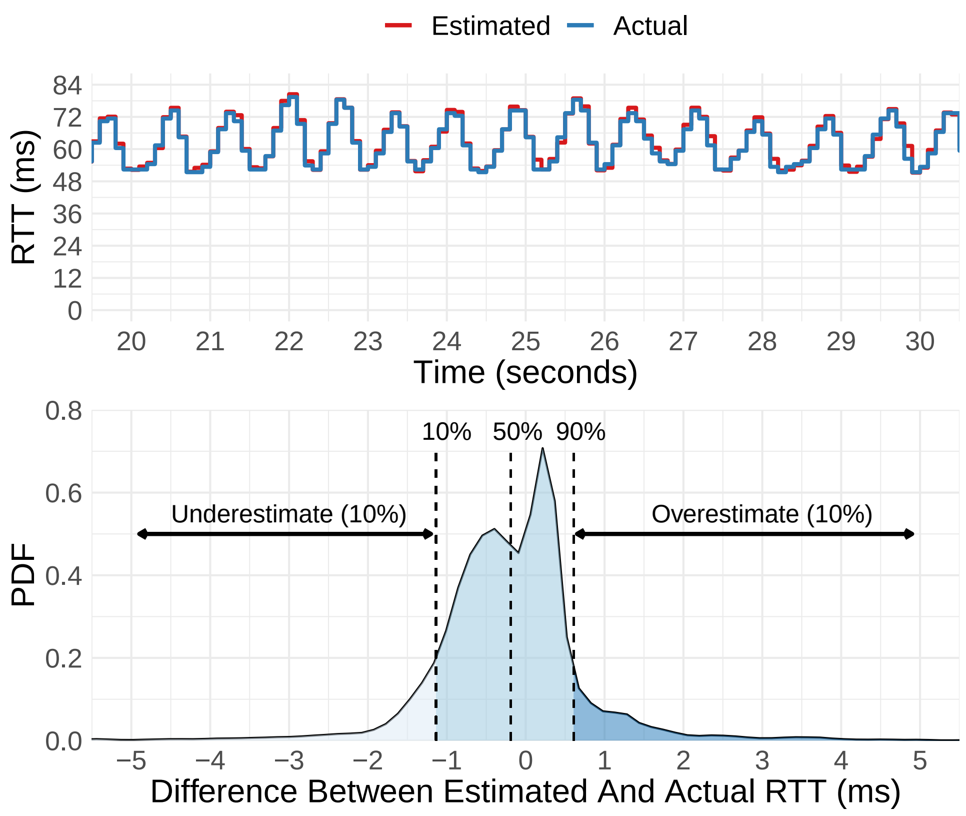


Figure 4.18: Bertha's estimate of the delay

message, the epoch size N is always rounded down to the nearest power of two. Doing this ensures that the epoch boundary packets sampled by the receivebox are either a strict superset or a strict subset of those sampled by the sendbox. The sendbox simply ignores the additional feedback messages in former case, and the recorded epoch boundaries for which no feedback has arrived in the latter.

Robust to packet loss. Note that our congestion measurement technique is robust to a boundary packet being lost between the sendbox and the receivebox. In this case, the sendbox would not get feedback for the lost boundary packet, and it would simply compute rates for the next boundary packet over a longer epoch once the next congestion ACK arrives.

Microbenchmarks. To evaluate the accuracy and robustness of this measurement technique, we picked 90 traces from our evaluation covering a range of link delays (20ms, 50ms, 100ms) and bottleneck rates (24Mbps, 48Mbps, 96Mbps), and computed the difference, at each time step, between Bundler's measurements (estimate) and the corresponding values measured at the bottleneck router (actual). In Figure 4.18 we focus on the RTT measurements: the bottom plot shows the distribution of the differences, and the top plot puts it into context by showing a five second segment from a trace where the median difference matched that of the full distribution. In Figure 4.17, we produce the same plots for the receive rate estimates. In summary, 80% of our RTT estimates were within 1.2ms of the actual value, and 80% of our receive rate estimates were within 4Mbps of the actual value.

Benefits

We use network emulation via mahimahi [95] to evaluate our implementation of Bundler in a controlled setting. There are three 8-core Ubuntu 18.04 machines in our emulated setup: (1) runs a sender, (2) runs a sendbox, and (3) runs both a receivebox and a receiver. We disable both TCP segmentation offload (TSO) and generic receive offload (GRO) as they would change the packet headers in between the sendbox and receivebox, which would cause inconsistent epoch boundary identification between the two boxes. Nevertheless, throughout our experiments CPU utilization on the machines remained below 10%.

Unless otherwise specified, we emulate the following scenario. A many-threaded client generates requests from a request size CDF drawn from an Internet core router [18] and assigns them to one of 200 server processes. The workload is heavy-tailed: 97.6% of requests are 10KB or shorter, and the largest 0.002% of requests are between 5MB and 100MB. Each server then sends the requested amount of data to the client and we measure the FCT of each such request. The link bandwidth at the mahimahi link is set to 96Mbps, and the RTT is set to 50ms. The requests result in an offered load of 84Mbps.

The endhost runs Cubic [54], and the sendbox runs Copa [7] with Nimbus [51] for cross traffic detection. The sendbox schedules traffic using the Linux kernel implementation of Stochastic Fairness Queueing (SFQ) [87]. Each experiment is comprised of 1,000,000 requests sampled from this distribution, across 10 runs each with a different random seed. We use median slowdown as our metric, where the "slowdown" of a request is its completion time divided by what its completion time would have been in an unloaded network. A slowdown of 1 is optimal, and lower numbers represent better performance.

We evaluate three configurations: (i) The "Status Quo" configuration represents the status quo: the sendbox simply forwards packets as it receives them, and the mahimahi bottleneck uses FIFO scheduling. (ii) The "In-Network" configuration deploys fair queueing at the mahimahi bottleneck.[9] Recall from §3.1 that this configuration is not deployable. (iii) The default Bundler configuration, that uses stochastic fair queueing [87] scheduling policy at the sendbox, and (iv) Using Bundler with FIFO (without exploiting scheduling opportunity).

[9]We implement this scheme by modifying mahimahi (our patch comprises 171 lines of C++) to add a packet-level fair-queueing scheduler to the bottleneck link.

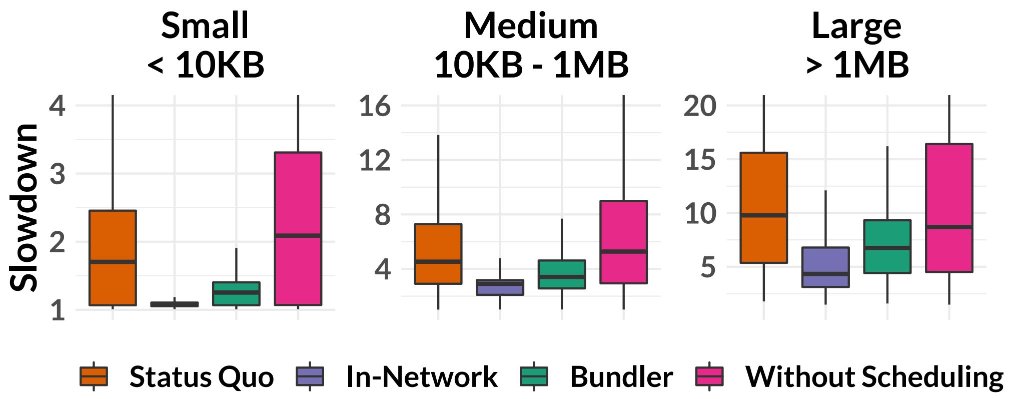


Figure 4.19: Bundler achieves 28% lower median slowdown. The three graphs show FCT distributions for the indicated request sizes: smaller than 10KB, between 10KB and 1MB, and greater than 1MB. Note the different y-axis scales for each group of request sizes. Whiskers show 1.25× the inter-quartile range. For both Bundler and In-Network, performance benefits come from preventing short flows from queueing behind long ones. Thus, Bundler's aggregate congestion control by itself is not enough; if we configure Bertha to use FIFO scheduling, the FCTs worsen compared to the status quo.

Figure 4.19 presents our results. The median slowdown (across all flow sizes) decreases from 1.76 for Baseline to 1.26 with Bundler 28% lower. In-Network's median slowdown is a further 15% lower then Bundler: 1.07. Meanwhile, in the tail, Bundler's 99%ile slowdown is 41.38, which is 48% lower than the Status Quo's 79.37. In-Network's 99%ile slowdown is 27.49.

4.8 Conclusion

We described the design, implementation, and evaluation of CCP, a system that restructures congestion control at the sender. CCP defines better abstractions for congestion control, specifying the responsibilities of the datapath and showing a way to use fold functions and control patterns to exercise control over datapath behavior. We showed how CCP (i) enables the same algorithm code to run on a variety of datapaths, (ii) increases the "velocity" of development and improves maintainability, and (iii) facilitates new capabilities such as the congestion manager-style aggregation and sophisticated signal processing algorithms.

Our implementation achieves high fidelity compared to native datapath implementations at low CPU overhead. The use of fold functions and summarization reduces overhead, but not at the expense of correctness or accuracy. We additionally showed multiple use cases of CCP, including in Park (§4.7.5) and Bundler (§4.7.6).

5 Conclusion

The systems, techniques, and abstractions this thesis has presented are generally not *necessary* for any individual application or network environment. It is entirely possible to engineer functional applications using traditional network stacks and ad-hoc communication libraries, just as it is entirely possible to re-implement congestion control algorithms for each datapath of interest. However, just as layered abstractions have helped us build these traditional tools to start with, this thesis has argued that new abstractions, techniques, and systems can help us scale our networked applications to meet ever-evolving demands of functionality, performance, and stability.

The fundamental reason new abstractions are necessary is the increasing amount of heterogeneity in our networks. Rather than provide one-size-fits-all networking abstractions, modern networks continue to adopt specialization as a means to provide greater efficiency and functionality. While traditional abstractions such as IP continue to be valuable, new abstractions such as the Chunnel as well as CCP's measurement and control primitives will allow applications and datapaths to embrace this increasing amount of heterogeneity. Bertha helps applications express the network features they want, which enables applications to decouple the specification of those features from their implementation. As a result, Bertha applications can defer implementation decisions to runtime, when information about the network runtime environment becomes available. Further, Bertha can provide stability and eliminate a class of bugs by checking implementations' compatibility during connection establishment. Meanwhile, CCP decouples congestion control algorithm implementations from the complexity of datapath runtime environments, and allows them to be re-used across datapaths.

Despite the progress Bertha and CCP offer, there remain opportunities for future work. One such direction is in understanding how datapath structures can help or hinder the performance of individual applications through the internal decisions they make. Given that Bertha can extend the traditional notion of the network stack upwards into what was earlier considered a part of the application's logic, how should we structure these stack components to best support the application's performance? Within the datapaths themselves, how can we support modular structures that acknowledge the new reality of datapath heterogeneity, and allow for component reuse while preserving performance?

It will always remain possible to engineer our way around heterogeneity with bespoke implementations and one-size-fits-all performance. Indeed, this method of problem solving is agile; developers can quickly build structures that adapt to contemporary trends in hardware or user demands. However, forever building ad-hoc structures eventually results in a loss of both application as well as developer efficiency. It will thus remain important to follow up with abstractions and structure that can provide all three of functionality, performance, and stability. This thesis is one step along this path.

Bibliography

[1] Acrimon. DashMap. https://github.com/xacrimon/dashmap. 3.6.4

[2] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data Center TCP (DCTCP). In *SIGCOMM*, 2010. 4.5.3

[3] Amazon. Amazon SNS. https://aws.amazon.com/sns/. 3.6.3

[4] Amazon. Amazon SQS Pricing. https://aws.amazon.com/sqs/pricing/. 3.6.3

[5] M. T. Arashloo, M. Ghobadi, J. Rexford, and D. Walker. HotCocoa: Hardware Congestion Control Abstractions. In *HotNets*, 2017. 4.2

[6] M. T. Arashloo, A. Lavrov, M. Ghobadi, J. Rexford, D. Walker, and D. Wentzlaff. Enabling Programmable Transport Protocols in High-Speed NICs . In *NSDI*, 2020. 2

[7] V. Arun and H. Balakrishnan. Copa: Congestion Control Combining Objective Optimization with Window Adjustments. In *NSDI*, 2018. 4.7.2, 4.7.6

[8] H. Balakrishnan, H. S. Rahul, and S. Seshan. An Integrated Congestion Management Architecture for Internet Hosts. In *SIGCOMM*, 1999. 4.1, 3, 4.2, 4.7.3

[9] Barroso, Luiz and Marty, Mike and Patterson, David and Ranganathan, Parthasarathy. Attack of the Killer Microseconds. *CACM*, 60(4):48–54, mar 2017. 2

[10] A. Belay, G. Prekas, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In *OSDI*, 2014. 2.1, 4.2

[11] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker. P4: Programming Protocol-independent Packet Processors. *SIGCOMM CCR*, 44(3):87–95, July 2014. 4.7.6

[12] R. Braden, L. Zhang, S. Berson, S. Herzog, and S. Jamin. Resource ReSerVation Protocol:(RSVP); Version 1 Functional Specification. 1997. 7

[13] L. Brakmo. TCP-BPF: Programmatically tuning TCP behavior through BPF. https://www.netdevconf.org/2.2/papers/brakmo-tcpbpf-talk.pdf. 4.2, 4.5.3

[14] L. S. Brakmo, S. W. O'Malley, and L. L. Peterson. TCP Vegas: New Techniques for Congestion Detection and Avoidance. In *SIGCOMM*, 1994. 4.7.5

[15] G. Cacheda. Google Pub/Sub performance (latency under low load)? `https://stackoverflow.com/a/53207646/1304393`, 2018. 3.6.3

[16] Q. Cai, S. Chaudhary, M. Vuppalapati, J. Hwang, and R. Agarwal. Understanding Host Network Stack Overheads. In *SIGCOMM*, 2021. 2.1

[17] CAIDA. The CAIDA Anonymized Internet Traces 2016 Dataset - 2016-01-21. `http://www.caida.org/data/passive/passive_2016_dataset.xml`, 2016. 4.6.1

[18] CAIDA. The CAIDA Anonymized Internet Traces 2016 Dataset - 2016-01-21. `http://www.caida.org/data/passive/passive_2016_dataset.xml`, 2016. 4.7.6

[19] F. Cangialosi*, A. Narayan*, P. Goyal, R. Mittal, M. Alizadeh, and H. Balakrishnan. Site-to-Site Internet Traffic Control. EuroSys, 2021. (document), 4.7.6

[20] N. Cardwell, Y. Cheng, C. S. Gunn, S. H. Yeganeh, and V. Jacobson. BBR: Congestion-Based Congestion Control. *ACM Queue*, 14(5), Oct. 2016. 4.4.3, 4.7.6

[21] M. J. Carey, D. J. DeWitt, D. Frank, G. Graefe, J. E. Richardson, E. J. Shekita, and M. Muralikrlshna. The Architecture of the EXODUS Extensible DBMS. In *On Object-Oriented Database Systems*, pages 231–256. Springer, 1991. 3.2

[22] M. Castro, M. Costa, J.-P. Martin, M. Peinado, P. Akritidis, A. Donnelly, P. Barham, and R. Black. Fast Byte-Granularity Software Fault Isolation. In *SOSP*, 2009. 4.3.1

[23] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *OSDI*, 2018. 3.2

[24] D. D. Clark, S. Shenker, and L. Zhang. Supporting Real-time Applications in an Integrated Services Packet Network: Architecture and Mechanism. In *SIGCOMM*, 1992. 4.7.6

[25] J. Corbet. Pluggable congestion avoidance modules. `https://lwn.net/Articles/128681/`, 2005. 4.2

[26] C. Craig. ISPs do throttle traffic – and the FCC can't stop it. `https://www.infoworld.com/article/2940538/internet/isps-do-throttle-traffic-and-the-fcc-cant-stop-it.html`, 2015. 4.7.6

[27] J. Daily, A. Vishnu, B. Palmer, H. Van Dam, and D. Kerbyson. On the suitability of MPI as a PGAS runtime. In *2014 21st International Conference on High Performance Computing (HiPC)*, pages 1–10. IEEE, 2014. 2

[28] A. Demers, S. Keshav, and S. Shenker. Analysis and Simulation of a Fair Queueing Algorithm. In *SIGCOMM*, 1989. 4.7.6

[29] A. Dhamdhere, D. Clark, A. Gamero-Garrido, M. Luckie, R. Mok, G. Akiwate, K. Gogia, V. Bajpai, A. Snoeren, and k. claffy. Inferring Persistent Interdomain Congestion. In *SIGCOMM*, 2018. 4.7.6

[30] M. Dong, Q. Li, D. Zarchy, P. B. Godfrey, and M. Schapira. PCC: Re-architecting Congestion Control for Consistent High Performance. In *NSDI*, 2015. 4.1

[31] M. Dong, T. Meng, D. Zarchy, E. Arslan, Y. Gilad, B. Godfrey, and M. Schapira. PCC Vivace: Online-Learning Congestion Control. In *NSDI*, 2018. 4.1

[32] DPDK. `http://dpdk.org/`. 2, 4.1

[33] A. Dragojević, D. Narayanan, E. B. Nightingale, M. Renzelmann, A. Shamis, A. Badam, and M. Castro. No compromises: Distributed transactions with consistency, availability, and performance. SOSP, 2015. 2

[34] N. Dukkipati, T. Refice, Y. Cheng, J. Chu, T. Herbert, A. Agarwal, A. Jain, and N. Sutin. An Argument for Increasing TCP's Initial Congestion Window. *SIGCOMM CCR*, 40(3), 2010. 2

[35] Linux Socket Filtering aka Berkeley Packet Filter (BPF). `https://www.kernel.org/doc/Documentation/networking/filter.txt`. 4.2, 4.5.3

[36] D. Ely, S. Savage, and D. Wetherall. Alpine: A user-level infrastructure for network protocol development. In *USITS*, volume 1, pages 15–15, 2001. 4.2

[37] H. Eran, L. Zeno, M. Tork, G. Malka, and M. Silberstein. NICA: An Infrastructure for Inline Acceleration of Network Applications. In *USENIX ATC*, 2019. 3.2

[38] U. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula. XFI: Software Guards for System Address Spaces. In *OSDI*, 2006. 4.3.1

[39] S. Floyd and V. Jacobson. Random Early Detection Gateways for Congestion Avoidance. *IEEE/ACM Trans. on Networking*, 1(4), Aug. 1993. 4.7.6

[40] Floyd, Sally. TCP and Explicit Congestion Notification. *CCR*, 24(5), Oct. 1994. 4.7.6

[41] B. Ford. Structured Streams: A New Transport Abstraction. In *SIGCOMM*, 2007. 4.2

[42] G. Fowler, L. C. Noll, K.-P. Vo, D. Eastlake, and T. Hansen. The FNV Non-Cryptographic Hash Algorithm. `https://tools.ietf.org/html/draft-eastlake-fnv-16`, 2018. 3.6.4

[43] D. Freimuth, E. Hu, J. LaVoie, R. Mraz, E. Nahum, P. Pradhan, and J. Tracey. Server Network Scalability and TCP Offload. In *USENIX ATC*, 2005. 3.2

[44] J. Fried, Z. Ruan, A. Ousterhout, and A. Belay. Caladan: Mitigating Interference at Microsecond Timescales. In *OSDI*, OSDI, 2020. 2.1, 2

[45] P. Garg and Y.-S. Wang. NVGRE: Network Virtualization Using Generic Routing Encapsulation, 2015. RFC 7637, IETF. 4.7.6

[46] Y. Ghigoff, J. Sopena, K. Lazri, A. Blin, and G. Muller. BMC: Accelerating Memcached using Safe In-kernel Caching and Pre-stack Processing. In *NSDI*, 2021. 3.6.4

[47] D. P. Ghormley, D. Petrou, S. H. Rodrigues, and T. E. Anderson. SLIC: An Extensibility System for Commodity Operating Systems. In *USENIX ATC*, 1998. 3.2

[48] Google. GCP PubSub Pricing. `https://cloud.google.com/pubsub/pricing`. 3.6.3

[49] Google. XLA: Optimizing Compiler for Machine Learning. `https://www.tensorflow.org/xla/`. 3.2

[50] P. Goyal, M. Alizadeh, and H. Balakrishnan. Rethinking Congestion Control for Cellular Networks. In *HotNets*, 2017. 4.5.3

[51] P. Goyal, A. Narayan, F. Cangialosi, D. Raghavan, S. Narayana, M. Alizadeh, and H. Balakrishnan. Elasticity Detection: A Building Block for Delay-Sensitive Congestion Control. *ArXiv e-prints*, Feb. 2018. (document), 4.1, 4.4.2, 4.7.1, 4.7.6

[52] G. Graefe. Volcano— An Extensible and Parallel Query Evaluation System. *IEEE Trans. on Knowl. and Data Eng.*, 6(1):120–135, Feb. 1994. 3.2

[53] H. S. Gunawi, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Deploying Safe User-level Network Services with icTCP. In *OSDI*, 2004. 4.2

[54] S. Ha, I. Rhee, and L. Xu. CUBIC: A New TCP-Friendly High-Speed TCP Variant. *SIGOPS Operating System Review*, July 2008. 4.6.1, 4.7.6, 4.7.6

[55] S. Han, K. Jang, A. Panda, S. Palkar, D. Han, and S. Ratnasamy. SoftNIC: A Software NIC to Augment Hardware. Technical Report UCB/EECS-2015-155, EECS Department, University of California, Berkeley, May 2015. 4.7.6

[56] S. Han, K. Jang, K. Park, and S. Moon. PacketShader: A GPU-Accelerated Software Router. In *SIGCOMM*, 2010. 2

[57] S. Han, S. Marshall, B.-G. Chun, and S. Ratnasamy. MegaPipe: A New Programming Interface for Scalable Network I/O. In *OSDI*, 2012. 2.1

[58] J. C. Hoe. Improving the Start-up Behavior of a Congestion Control Scheme for TCP. In *SIGCOMM*, 1996. 4.6.1

[59] D. Hudak. Introduction to the Partitioned Global Address Space (PGAS) Programming Model. https://www.osc.edu/sites/osc.edu/files/staff_files/dhudak/pgas-tutorial.pdf. 2

[60] N. C. Hutchinson and L. L. Peterson. The x-kernel: An architecture for implementing network protocols. *IEEE Trans. Softw. Eng.*, 17(1), 1991. 3.2

[61] S. Ibanez, A. Mallery, S. Arslan, T. Jepsen, M. Shahbaz, C. Kim, and N. McKeown. The nanoPU: A Nanosecond Network Stack for Datacenters. In *OSDI*, 2021. 2

[62] V. Jacobson. Congestion Avoidance and Control. In *SIGCOMM*, 1988. 4.7.6

[63] E. Jeong, S. Wood, M. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park. mTCP: a Highly Scalable User-level TCP Stack for Multicore Systems. In *NSDI*, 2014. 2, 4.1

[64] Z. Jia, M. Zaharia, and A. Aiken. Beyond Data and Model Parallelism for Deep Neural Networks. In *MLSys*, 2019. 3.2

[65] X. Jin, X. Li, H. Zhang, R. Soulé, J. Lee, N. Foster, C. Kim, and I. Stoica. NetCache: Balancing Key-Value Stores with Fast In-Network Caching. In *SOSP*, 2017. 3.6.4

[66] A. B. Johnston and D. C. Burnett. *WebRTC: APIs and RTCWEB Protocols of the HTML5 Real-time Web*. Digital Codex LLC, 2012. 4.1

[67] N. Kabra and D. J. DeWitt. OPT++: An Object-Oriented Implementation for Extensible Database Query Optimization. *The VLDB Journal*, 1999. 3.2

[68] A. Kalia, M. Kaminsky, and D. G. Andersen. Using RDMA Efficiently for Key-Value Services. In *SIGCOMM*, 2014. 2

[69] A. Kalia, M. Kaminsky, and D. G. Andersen. Datacenter rpcs can be general and fast. NSDI, 2019. 2

[70] D. Kaloper-Mersinjak, H. Mehnert, A. Madhavapeddy, and P. Sewell. Not-quite-so-broken tls: Lessons in re-engineering a security protocol specification and implementation. In *USENIX Security*, 2015. 3.1

[71] A. Kaufmann, T. Stamler, S. Peter, N. K. Sharma, A. Krishnamurthy, and T. Anderson. TAS: TCP Acceleration as an OS Service. EuroSys, 2019. 2

[72] E. Kohler, R. Morris, and B. Chen. Programming Language Optimizations for Modular Router Configurations. In *ASPLOS*, 2002. 3.2

[73] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click modular router. *ACM Transactions on Computer Systems (TOCS)*, 18(3):263–297, 2000. 3.2

[74] B. Kovalev and P. Rudenko. RDG: Accelerating Apache Spark 3.0 with RAPIDS Accelerator over RoCE network. `https://docs.mellanox.com/pages/releaseview.action?pageId=28938181`, 2020. 3.6.4

[75] J. Kreps, N. Narkhede, J. Rao, et al. Kafka: A Distributed Messaging System for Log Processing. In *NetDB*, volume 11, pages 1–7, 2011. 3.6.3

[76] A. Langley, A. Riddoch, A. Wilk, A. Vicente, C. Krasic, D. Zhang, F. Yang, F. Kouranov, I. Swett, J. Iyengar, J. Bailey, J. Dorfman, J. Roskind, J. Kulik, P. Westin, R. Tenneti, R. Shade, R. Hamilton, V. Vasiliev, W.-T. Chang, and Z. Shi. The QUIC Transport Protocol: Design and Internet-Scale Deployment. In *SIGCOMM*, 2017. 2.1, 3.5.1, 4.1, 4.2, 4.7.6

[77] F. Li, A. A. Niaki, D. Choffnes, P. Gill, and A. Mislove. A Large-Scale Analysis of Deployed Traffic Differentiation Practices. In *SIGCOMM*, 2019. 4.7.6

[78] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky. MICA: A Holistic Approach to Fast in-Memory Key-Value Storage. In *NSDI*, 2014. 2

[79] LLVM Authors. LLVM. `https://llvm.org/docs/LangRef.html`. 3.2

[80] M. Mahalingam, D. Dutt, K. Duda, P. Agarwal, L. Kreeger, T. Sridhar, M. Bursell, and C. Wright. Virtual eXtensible Local Area Network (VXLAN): A Framework for Overlaying Virtualized Layer 2 Networks over Layer 3 Networks, 2014. RFC 7648, IETF. 4.7.6

[81] H. Mao, P. Negi, A. Narayan, H. Wang, J. Yang, H. Wang, R. Marcus, R. Addanki, M. K. Shirkoohi, S. He, V. Nathan, F. Cangialosi, S. Venkatakrishnan, W.-H. Weng, S. Han, T. Kraska, and M. Alizadeh. Park: An Open Platform for Learning-Augmented Computer Systems. In *NeurIPS*, 2019. (document), 4.7.5

[82] Y. Mao, H. Chen, D. Zhou, X. Wang, N. Zeldovich, and M. F. Kaashoek. Software Fault Isolation with API Integrity and Multi-Principal Modules. In *SOSP*, 2011. 4.3.1

[83] M. Marty, M. de Kruijf, J. Adriaens, C. Alfeld, S. Bauer, C. Contavalli, M. Dalton, N. Dukkipati, W. C. Evans, S. Gribble, N. Kidd, R. Kononov, G. Kumar, C. Mauer, E. Musick, L. Olson, E. Rubow, M. Ryan, K. Springborn, P. Turner, V. Valancius, X. Wang, and A. Vahdat. Snap: A Microkernel Approach to Host Networking. In *SOSP*, 2019. 2.1

[84] J. McCauley, M. Zhao, E. Jackson, B. Raghavan, S. Ratnasamy, and S. Shenker. Taking an AXE to L2 Spanning Trees. In *SIGCOMM*, 2016. 4.7.6

[85] McCauley, James and Harchol, Yotam and Panda, Aurojit and Raghavan, Barath and Shenker, Scott. Enabling a Permanent Revolution in Internet Architecture. In *SIGCOMM*, 2019. 4.7.6

[86] W. J. McKenna, L. Burger, C. Hoang, and M. Truong. EROC: a toolkit for building NEATO query optimizers. In *VLDB*, 1996. 3.2

[87] McKenney, Paul E. Stochastic Fairness Queueing. In *INFOCOM*, 1990. 4.7.6, 4.7.6, 4.7.6

[88] Microsoft. Azure Queues Storage Pricing. `https://azure.microsoft.com/en-us/pricing/details/storage/queues/`. 3.6.3

[89] Microsoft. Azure Service Bus Pricing. `https://azure.microsoft.com/en-us/pricing/details/service-bus/`. 3.6.3

[90] Microsoft. The Official Microsoft IIS Site. `https://www.iis.net/`. 3.2

[91] C. Mitchell, Y. Geng, and J. Li. Using One-Sided RDMA Reads to Build a Fast, CPU-Efficient Key-Value Store. In *USENIX ATC*, 2013. 2

[92] P. Mochel. The Linux Kernel Device Model. In *Ottawa Linux Symposium*, page 368, 2002. 3.2

[93] A. Narayan, F. Cangialosi, D. Raghavan, P. Goyal, S. Narayana, R. Mittal, M. Alizadeh, and H. Balakrishnan. Restructuring endpoint congestion control. SIGCOMM, 2018. (document), 4.7.6, 4.7.6

[94] A. Narayan, A. Panda, M. Alizadeh, H. Balakrishnan, A. Krishnamurthy, and S. Shenker. Bertha: Tunneling through the Network API. HotNets, 2020. (document)

[95] R. Netravali, A. Sivaraman, S. Das, A. Goyal, K. Winstein, J. Mickens, and H. Balakrishnan. Mahimahi: Accurate Record-and-Replay for HTTP. In *USENIX ATC*, 2015. 4.6.1, 4.6.1, 4.7.6

[96] Netronome. Agilio LX SmartNICs. `https://www.netronome.com/products/agilio-lx/`. [Online, Retrieved July 28, 2017]. 4.1

[97] K. Nichols and V. Jacobson. Controlling Queue Delay. *ACM Queue*, 10(5), May 2012. 4.7.6

[98] E. Nordström, D. Shue, P. Gopalan, R. Kiefer, M. Arye, S. Y. Ko, J. Rexford, and M. J. Freedman. Serval: An end-host stack for service-centric networking. In *NSDI*, 2012. 3.2

[99] The Network Simulator. `https://www.isi.edu/nsnam/ns/`. 4.6.3

[100] ONNX. Open Neural Network Exchange. `https://onnx.ai/`. 3.2

[101] OpenAI Gym. `https://gym.openai.com/`. 4.7.5

[102] A. Ousterhout, J. Fried, J. Behrens, A. Belay, and H. Balakrishnan. Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads. In *NSDI*, 2019. 2.1, 3.3.1, 3.4.1, 3.5.2

[103] T. Overby, F. Mazzoli, D. Tolnay, and Z. Riordan. Bincode. `https://github.com/bincode-org/bincode`. 3.6.3

[104] S. Palkar, J. Thomas, D. Narayanan, P. Thaker, R. Palamuttam, P. Negi, A. Shanbhag, M. Schwarzkopf, H. Pirk, S. Amarasinghe, S. Madden, and M. Zaharia. Evaluating End-to-End Optimization for Data Analytics Applications in Weld. In *VLDB*, 2018. 3.2

[105] R. Pan, P. Natarajan, C. Piglione, M. Prabhu, V. Subramanian, F. Baker, and B. VerSteeg. PIE: A lightweight control scheme to address the bufferbloat problem. In *HPSR*, 2013. 4.7.6

[106] C. Perkins. IP Encapsulation within IP, 1996. RFC 2003, IETF. 4.7.6

[107] S. Peter, J. Li, I. Zhang, D. R. K. Ports, D. Woos, A. Krishnamurthy, T. Anderson, and T. Roscoe. Arrakis: The Operating System is the Control Plane. In *OSDI*, 2014. 4.2

[108] P. M. Phothilimthana, M. Liu, A. Kaufmann, S. Peter, R. Bodik, and T. Anderson. Floem: A Programming System for NIC-Accelerated Network Applications. In *OSDI*, 2018. 2

[109] B. Pismenny, H. Eran, A. Yehezkel, L. Liss, A. Morrison, and D. Tsafrir. Autonomous NIC offloads. In *ASPLOS*, 2021. 3.2

[110] G. Prekas, M. Kogias, and E. Bugnion. Zygos: Achieving low tail latency for microsecond-scale networked tasks. SOSP, 2017. 2.1

[111] N. Provos. Improving host security with system call policies. In *Proceedings of the 12th Usenix Security Symposium*, 2002. 4.3.1

[112] PureVPN. ISP Bandwidth Throttling Explained. `https://www.purevpn.com/blog/isp-bandwidth-throttling-explained/`, 2017. 4.7.6

[113] D. Ritchie. A Stream Input-Output System. *AT&T Bell Laboratories Technical Journal*, 63, 1984. 4.6.2

[114] L. Rizzo. netmap: A Novel Framework for Fast Packet I/O. In *USENIX ATC*, 2012. 2, 4.1

[115] rustfft. `https://crates.io/crates/rustfft`. 4.7.1

[116] S. Blake and D. Black and M. Carlson and E. Davies and Z. Wang and W. Weiss. An Architecture for Differentiated Services. RFC 2475, 1998. 4.7.6

[117] J. Salim, H. Khosravi, A. Kleen, and A. Kuznetsov. Linux Netlink as an IP Services Protocol, 2003. RFC 3819, IETF. 4.6.2

[118] Savage, Stefan and Wetherall, David and Karlin, Anna and Anderson, Tom. Practical Network Support for IP Traceback. In *SIGCOMM*, 2000. 4.7.6

[119] H. Shan, N. J. Wright, J. Shalf, K. Yelick, M. Wagner, and N. Wichmann. A preliminary evaluation of the hardware acceleration of the Cray Gemini interconnect for PGAS languages and comparison with MPI. *ACM SIGMETRICS Performance Evaluation Review*, 40(2):92–98, 2012. 2

[120] J. Sherry, D. Kim, S. Mahalingam, A. Tang, S. Wang, and S. Ratnasamy. Netcalls: End Host Function Calls to Network Traffic Processing Services. UC Berkeley Technical Report

No. UCB/EECS-2012-175. `http://www.eecs.berkeley.edu/Pubs/TechRpts/2012/EECS-2012-175.html`, 2012. 3.2

[121] M. Shreedhar and G. Varghese. Efficient fair queueing using deficit round robin. In *SIGCOMM*, 1995. 4.7.6

[122] M. Sipser. *Introduction to the Theory of Computation*. Course Technology, third edition, 2013. 3.4.1

[123] C. Staub. Ghostunnel. `https://github.com/ghostunnel/ghostunnel`. 3.6.2

[124] I. Stoica, S. Shenker, and H. Zhang. Core-stateless Fair Queueing: A Scalable Architecture to Approximate Fair Bandwidth Allocations in High-speed Networks. *IEEE/ACM Trans. Netw.*, 2003. 4.7.6

[125] D. L. Tennenhouse and D. J. Wetherall. Towards an Active Network Architecture. *SIGCOMM CCR*, October 2007. 3.2

[126] I. Thomas, I. Goldberg, D. Wagner, R. Thomas, and E. Brewer. A Secure Environment for Untrusted Helper Applications (Confining the Wily Hacker). In *Proceedings of the 6th conference on USENIX Security Symposium*, 2001. 4.3.1

[127] M. Tirmazi, R. Ben Basat, J. Gao, and M. Yu. Cheetah: Accelerating Database Queries with Switch Pruning. In *SIGMOD*, 2020. 3.6.4

[128] A. Tirumala, F. Qin, J. Dugan, J. Ferguson, and K. Gibbs. Iperf: The TCP/UDP Bandwidth Measurement Tool. *http://dast.nlanr.net/Projects*, 2005. 4.6.2

[129] V. Tran and O. Bonaventure. Beyond socket options: making the Linux TCP stack truly extensible. *CoRR*, 2019. 3.2

[130] V.-H. Tran and O. Bonaventure. Making the Linux TCP Stack More Extensible With eBPF. In *Netdev 0x13*, 2019. 3.2

[131] C.-C. Tu, J. Stringer, and J. Pettit. Building an Extensible Open VSwitch Datapath. *SIGOPS Oper. Syst. Rev.*, page 72–77, Sept. 2017. 3.2

[132] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient Software-based Fault Isolation. In *SOSP*, 1993. 4.3.1

[133] M. Walfish, J. Stribling, M. N. Krohn, H. Balakrishnan, R. T. Morris, and S. Shenker. Middleboxes No Longer Considered Harmful. In *OSDI*, 2004. 3.2

[134] K. Winstein and H. Balakrishnan. Mosh: An Interactive Remote Shell for Mobile Clients. In *USENIX Annual Technical Conference*, 2012. 4.1

[135] K. Winstein and H. Balakrishnan. TCP ex Machina: Computer-Generated Congestion Control. In *SIGCOMM*, 2013. 4.1

[136] K. Winstein, A. Sivaraman, and H. Balakrishnan. Stochastic Forecasts Achieve High Throughput and Low Delay over Cellular Networks. In *NSDI*, 2013. 4.1

[137] D. Wischik, C. Raiciu, A. Greenhalgh, and M. Handley. Design, Implementation and Evaluation of Congestion Control for Multipath TCP. In *NSDI*, 2011. 4.1

[138] J. Wroclawski et al. The Use of RSVP with IETF Integrated Services, 1997. 7

[139] F. Y. Yan, J. Ma, G. Hill, D. Raghavan, R. S. Wahby, P. Levis, and K. Winstein. Pantheon: The Training Ground for Internet Congestion-Control research. In *USENIX ATC*, 2018. 4.1

[140] B. Yee, D. Sehr, G. Dardyk, B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native Client: A Sandbox for Portable, Untrusted x86 Native Code. In *IEEE Symposium on Security and Privacy*, 2009. 4.3.1

[141] I. Zhang, J. Liu, A. Austin, M. L. Roberts, and A. Badam. I'm Not Dead Yet! The Role of the Operating System in a Kernel-Bypass Era. In *HotOS*, 2019. 2.1

[142] L. Zhang. Virtual Clock: A New Traffic Control Algorithm for Packet Switching Networks. In *SIGCOMM*, 1990. 4.7.6

[143] Y. Zhu, H. Eran, D. Firestone, C. Guo, M. Lipshteyn, Y. Liron, J. Padhye, S. Raindel, M. H. Yahia, and M. Zhang. Congestion Control for Large-Scale RDMA Deployments. In *SIGCOMM*, 2015. 4.1

[144] D. Zhuo, K. Zhang, Y. Zhu, H. H. Liu, M. Rockett, A. Krishnamurthy, and T. Anderson. Slim: OS Kernel Support for a Low-Overhead Container Overlay Network. In *NSDI*, 2019. 3.6.2