Justitia: Software Multi-Tenancy in Hardware Kernel-Bypass Networks

Yiwen Zhang, *University of Michigan;* Yue Tan, *University of Michigan and Princeton University;* Brent Stephens, *University of Illinois at Chicago;* Mosharaf Chowdhury, *University of Michigan*

https://www.usenix.org/conference/nsdi22/presentation/zhang-yiwen

This paper is included in the Proceedings of the 19th USENIX Symposium on Networked Systems Design and Implementation.

April 4–6, 2022 • Renton, WA, USA

978-1-939133-27-4

Open access to the Proceedings of the 19th USENIX Symposium on Networked Systems Design and Implementation is sponsored by

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

Justitia: Software Multi-Tenancy in Hardware Kernel-Bypass Networks

Yiwen Zhang*, Yue Tan*◇, Brent Stephens†, and Mosharaf Chowdhury*
*University of Michigan, ◇Princeton University, †University of Illinois at Chicago

Abstract

Kernel-bypass networking (KBN) is becoming the new norm in modern datacenters. While hardware-based KBN offloads all dataplane tasks to specialized NICs to achieve better latency and CPU efficiency than software-based KBN, it also takes away the operator's control over network sharing policies. Providing policy support in multi-tenant hardware KBN brings unique challenges – namely, preserving ultra-low latency and low CPU cost, finding a well-defined point of mediation, and rethinking traffic shapers. We present Justitia to address these challenges with three key design aspects: (i) Split Connection with message-level shaping, (ii) sender-based resource mediation together with receiver-side updates, and (iii) passive latency monitoring. Using a latency target as its knob, Justitia enables multi-tenancy policies such as predictable latencies and fair/weighted resource sharing. Our evaluation shows Justitia can effectively isolate latency-sensitive applications at the cost of slightly decreased utilization and ensure that throughput and bandwidth of the rest are not unfairly penalized.

1 Introduction

To deal with the growing demands of ultra-low latency with high throughput (message rates) and high bandwidth in large fan-out services, ranging from parallel lookups in in-memory caches [16, 30, 32] and resource disaggregation [2, 22, 52] to analytics and machine learning [1, 26, 47], kernel-bypass networking (KBN) is becoming the new norm in modern datacenters [14, 23, 43, 44, 64]. As the name suggests, with KBN, applications bypass the operating system (OS) kernel to improve performance while relieving the CPU.

There are two major trends in KBN today. *Software-based KBN* (e.g., DPDK) removes the kernel from the data path and performs packet processing in the user space. In contrast, *hardware-based KBN* (e.g., RDMA) further lowers latency by at least one order of magnitude and reduces CPU usage by offloading dataplane tasks to specialized NICs (e.g., RDMA NICs) with on-board compute.

Hardware KBN, however, takes away the operator's control over network sharing policies such as prioritization, isolation, and performance guarantees. Unlike software KBN, coexisting applications must rely on the specialized NIC to arbitrate among data transfer operations once they are posted to the hardware. We observe that existing hardware KBNs provide poor support for multi-tenancy. For example, even for real-world applications such as DARE [49], eRPC [32], and FaSST [31], sharing the same NIC leads to severe performance anomalies including unpredictable latency, throttled throughput (i.e., lower message rates), and unfair bandwidth sharing (§3). In this paper, we aim to address the following question: *Can we marry the benefits of software KBN with the efficiency of hardware KBN and enable fine-grained multi-tenancy support?*

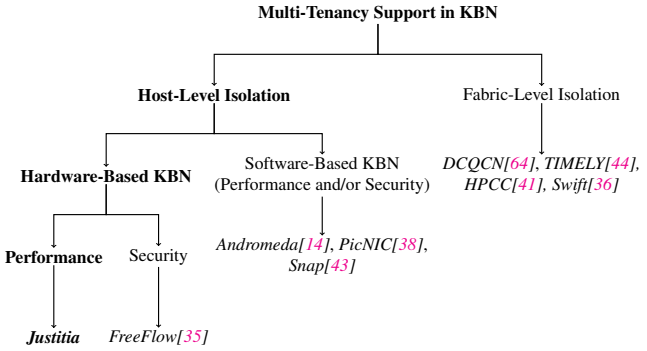


Figure 1: ***Design space for multi-tenancy support in KBN.***

Recent works have explored multi-tenancy support in large-scale software-based KBN deployments [14, 38, 43]. Their designs enforce fine-grained sharing policies such as performance and security (address space) isolation at the end hosts and pair with fabric-level solutions (e.g., congestion control) in case the network fabric becomes a bottleneck (Figure 1). However, existing software KBN solutions cannot be applied to hardware-based KBN due to three unique challenges:

1. Because host CPU is no longer involved, common CPU-based resource allocation mechanism cannot be applied. Instead, tenants issue RDMA operations with arbitrary data load at no CPU cost, which leaves no obvious point of control to exert resource mediation.
2. Hardware offloading brings packetization from user space into the NIC, disabling fine-grained user-space shaping at the packet level [27, 51].
3. It is also crucial to preserve hardware-based KBN's efficiency (i.e., single μs latency and low CPU cost[1]) while providing multi-tenancy support.

We present Justitia, a software-only solution that enables

[1]This does not apply to applications that aim for low latency or high message rates and busy spin their cores for maximum performance.

multi-tenancy support in hardware-based KBN, to address the aforementioned challenges (§4). Our key idea is to introduce an efficient software mediator in front of the NIC that can implement performance-related multi-tenancy policies – including (1) fair/weighted resource sharing and (2) predictable latencies while maximizing utilization or a mix of the two. Given that RDMA is the primary hardware-based KBN implementation today, in this paper, we specifically focus our solutions on RDMA NICs (RNICs).

Enabling fine-grained sharing policies in RDMA requires an efficient way of managing RNIC resources (i.e., link bandwidth and execution throughput). To this end, we propose *Split Connections* that decouple a tenant application's intent from its actuation and introduces a point of resource mediation. Justitia mediates RNIC resources by combining the benefits of sender-based and receiver-based design. RDMA operations are split and paced at the sender side before placing them onto the RNIC; receiver-side updates are collected to avoid spurious resource allocation caused by either incast or RDMA READ contention. Shaping is performed at the message level, where message sizes and their pacing rate are adjusted dynamically based on the current policy in use. By splitting RDMA connections, Justitia can effectively manage tenants' connections to consume RNIC resources based on the policy we set instead of letting tenants themselves compete by arbitrarily issuing RDMA operations.

To provide predictable latencies for latency-sensitive applications, Justitia introduces the concept of *reference flow* and monitors its latency instead of intercepting low-latency tenant applications. By comparing the latency measurements of many reference flows from the same sender machine to different receivers, Justitia can quickly detect (local and remote) RNIC resource contention. Given a tail latency target, Justitia maximizes RNIC resource utilization without violating the target. When the target is unachievable, based on the operator-defined policy, Justitia can choose to ensure that each of the competing n entities gets at least $\frac{1}{n}$th of one of the RNIC's two resources, extending the classic hose model of network sharing [17] to multi-resource RNICs.

We have implemented (§5) and evaluated (§6) Justitia on both InfiniBand and RoCEv2 networks. It provides multi-tenancy support among different types of applications without incurring high CPU usage (1 CPU core per host), introducing additional overheads, or modifying application codes. For example, using Justitia, DARE's tail latency improves by 3.4× when running in parallel with Apache Crail [5, 60], a bandwidth-sensitive storage application, and Justitia preserves 81% of Crail's original performance. Justitia also complements RDMA congestion control protocols like DCQCN [64] while further mitigating receiver-side RNIC contention, and reduces tail latency even when the network is congested.

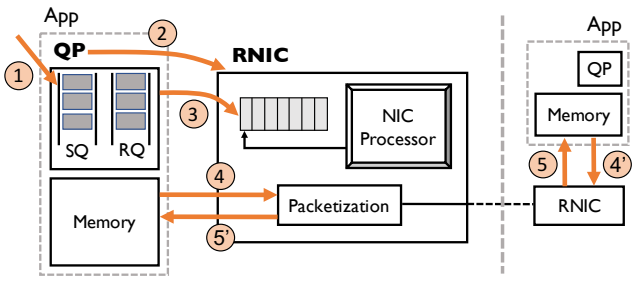


Figure 2: ***Overview of host-RNIC interaction when posting (i) an RDMA WRITE operation (① → ②. → ③ → ④ → ⑤) and (ii) an RDMA READ operation (① → ② → ③ → ④' → ⑤').***

2 Background

Recent works [36, 38] have discovered unpredictable latencies due to end-host resource contention, but their primary focus is on receiver-side engine congestion in software-based KBN. In this work, we aim to emphasize that *sender-side resource contention* in hardware-based KBN such as RDMA can also lead to severe performance degradation when multiple tenants coexist. An ideal solution should address both sender- and receiver-side issues. In this section, we give an overview on how an RDMA operation is performed, followed by the root cause of RDMA's lack of multi-tenancy support.

2.1 Life Cycle of an RDMA Operation

RDMA enables direct access between user-registered memory regions without involving the OS kernel, offloading data transfer tasks to the RNIC. Applications initiate RDMA operations by posting Work Requests (WRs) via Queue Pairs (QPs) to describe the messages to transmit. Figure 2 shows how an RDMA application interacts with an RNIC to initiate an RDMA operation. To start an RDMA WRITE, ① the user application place a Work Queue Element (WQE) describing the message to the Send Queue (SQ), and ② rings a door bell to notify the RNIC by writing its QP number into the corresponding doorbell register on RNIC. At this point, the user application has completed its task and offloads the rest of the work to RNIC. After the RNIC gets notified, it ③ fetches and processes the requests from the send queue, and ④ pulls the message from the user memory, splits it into packets, and sends it to the remote RNIC. Finally, the remote RNIC ⑤ writes the received message directly into the remote memory.

In the case of an RDMA READ operation, the user application again posts the WQE and notifies the RNIC to collect it (① → ③). The local RNIC then ④' notifies the remote RNIC to pull the data from remote memory, and ⑤' places the message back to local memory after de-packetizing the received packets. Despite the opposite direction of data transfer, the remote OS remains passive just as the case with an RDMA WRITE. In both cases, the sender of the RDMA operation actively controls what goes into the RNIC while the remote side stays passively unaware.[2]

[2]This is true even for two-sided operations that require the receiver to post WQEs to its Receive Queue before a Send Request arrives. We still

2.2 Lack of Multi-Tenancy Support

RDMA lacks multi-tenancy support for two primary reasons: (i) tenants/applications compete for multiple RNIC resources, and (ii) RNIC processes ready-to-consume message in a greedy fashion to maximize utilization. Both are related to different symptoms of the isolation issues.

Multi-Resource Contention There exist two primary resources that need to be shared on an RNIC: *link bandwidth* and *execution throughput*. Bandwidth-sensitive applications consume RNIC's link bandwidth to issue large DMA requests. Throughput-sensitive applications, on the other hand, consume RNIC's execution throughput to issue small DMA requests in batches. Latency-sensitive applications, however, consume neither resource with the small messages they sparsely send. As we will soon show (§3), isolation anomalies can occur when applications compete for different resources.

Greedy Processing for High Utilization Although the actual RNIC implementation details are private, we can consider two hypotheses on how RNIC handles multiple requests simultaneously: either the RNIC buffers WQEs collected in ③ in Figure 2 from multiple applications and arbitrates among them using some scheduling mechanism; or it processes them in a greedy manner. When a latency-sensitive application competes with a bandwidth-sensitive application, too much arbitration in the former can cause low resource utilization (e.g., unable to catch up the line rate), whereas too little arbitration in the latter leads to head-of-line (HOL) blocking (which leads to latency variation). Our observations across all three RDMA implementations (§3), where applications using small messages are consistently affected by the ones using larger ones, suggest the latter. Note that even though receiver-side congestion can also happen during step ⑤ as pointed out in [38], both root causes can easily stem from the sender side of the operation via step ③ and thus cannot be ignored. We elaborate on how Justitia mitigates both sender- and receiver-side issues in Section 4.

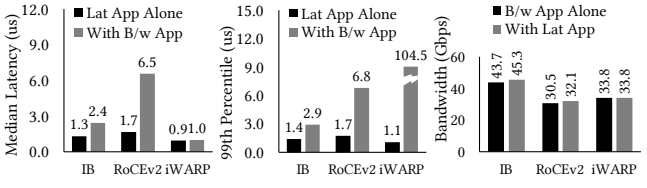
3 Performance Isolation Anomalies in RDMA

This section establishes a baseline understanding of sharing characteristics in hardware KBN and identifies common isolation anomalies across different RDMA implementations with both microbenchmarks (§3.1) and highly optimized, state-of-the-art RDMA-based applications (§3.2).

To study RDMA sharing characteristics among applications with different objectives, we consider three major types of RDMA-enabled applications:

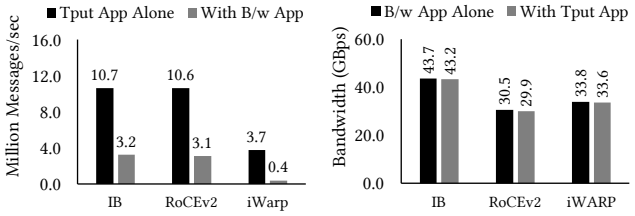
1. *Latency-Sensitive:* Sends small messages and cares about the individual message latencies.
2. *Throughput-Sensitive:* Sends small messages in batches to maximize the number of messages sent per second.

consider the receiver as passive because it can only control where to place a message but cannot control *when* a message will arrive.



(a) *Latency App (Med)* (b) *Latency App (99th)* (c) *Bandwidth App*

Figure 3: ***Latency-sensitive applications require isolation against bandwidth-sensitive applications.***



(a) *Throughput App* (b) *Bandwidth App*

Figure 4: ***Throughput-sensitive application requires isolation from bandwidth-sensitive applications.***

3. *Bandwidth-Sensitive:* Sends large messages with high bandwidth requirements.

Summary of Key Findings:

- Both latency- and throughput-sensitive applications need isolation from bandwidth-sensitive applications (§3.1.1).
- If only latency- or throughput-sensitive applications (or a mix of the two types) compete, they are isolated from each other (§3.1.2).
- Multiple bandwidth-sensitive applications can lead to unfair bandwidth allocations depending on their message sizes (§3.1.3).
- Highly optimized, state-of-the-art RDMA-based systems also suffer from the anomalies we discovered (§3.2).

In the rest of this section, we describe our experimental settings and elaborate on these findings.

3.1 Observations From Microbenchmarks

We performed microbenchmarks between two machines with the same type of RNIC, where both are connected to the same RDMA-enabled switch. For most of the experiments, we used 56 Gbps Mellanox ConnectX-3 Pro for InfiniBand, 40 Gbps Mellanox ConnectX-4 for RoCEv2, and 40 Gbps Chelsio T62100 for iWARP; 10 and 100 Gbps settings are described similar. More details on our hardware setups are in Table 1 of Appendix A.

Our benchmarking applications are written based on Mellanox perftest [61] and each of them uses a single Queue Pair. Unless otherwise specified, latency-sensitive applications in our microbenchmarks send a continuous stream of 16B messages, throughput-sensitive ones send a continuous stream of batches with each batch having 64 16B messages, and bandwidth-sensitive applications send a continuous stream of 1MB messages. Although all applications send messages

using RDMA WRITEs over reliable connection (RC) QPs in the observations below, other verbs show similar anomalies as well. We defer the usage and discussion of hardware virtual lanes to Section 6.3.

3.1.1 Both Latency- and Throughput-Sensitive Applications Require Isolation

The performance of the latency-sensitive applications deteriorate for all RDMA implementations (Figure 3). Out of the three implementations we benchmarked, InfiniBand and RoCEv2 observes 1.85× and 3.82× degradations in median latency and 2.23× and 4× at the 99th percentile. While iWARP performs well in terms of median latency, its tail latency degrades dramatically (95×).

Throughput-sensitive applications also suffer. When a background bandwidth-sensitive application is running, the throughput-sensitive ones observe a throughput drop of 2.85× or more across all RDMA implementations (Figure 4). Note that in our microbenchmark with 1 QP per application, throughput-sensitive applications that consume NIC execution throughput hit the bottleneck. This does not imply RNIC always favors link bandwidth over execution throughput. We notice RNIC bandwidth starts to become the bottleneck when there exists 4× more throughput-sensitive applications.

More importantly, both latency- and throughput-sensitive applications experience more severe performance degradations (e.g., 139X worse latency with the presence of 16 bandwidth applications) as more bandwidth-sensitive applications join the competition, which is prevalent in shared datacenters [23, 64]. Appendix B.1 provides more details.

3.1.2 Latency-Sensitive Applications Coexist Well; So Do Throughput-Sensitive Ones

We observe no obvious anomalies among latency- or throughput-sensitive applications, or a mix of the two types. Detailed results can be found in Appendix B.

3.1.3 Bandwidth-Sensitive Applications Hurt Each Other

Unlike latency- and throughput-sensitive applications, bandwidth-sensitive applications with different message sizes do affect each other.Figure 5 shows that a bandwidth-sensitive application using 1MB messages receive smaller share than one using 1GB messages. The latter receives 1.42×, 1.22× and 1.51× more bandwidth in InfiniBand, RoCEv2, and iWARP, respectively.

3.1.4 Anomalies are Present in Faster Networks Too

We performed the same benchmarks on 100 Gbps InfiniBand, only to observe that most of the aforementioned anomalies are still present. Appendix C.1 has the details.

3.2 Isolation Among Real-World Applications

In this section, we demonstrate how real RDMA-based systems fail to preserve their performance in the presence of the aforementioned anomalies.

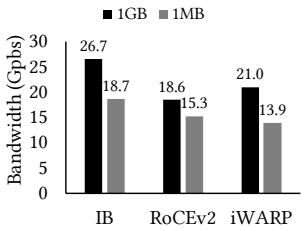


Figure 5: ***Anomalies among Bandwidth-sensitive applications with different message sizes.***

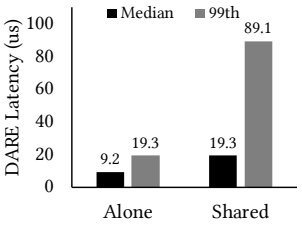


Figure 6: ***Latency of DARE's Put and Get operations when coexisting with Apache Crail's storage traffic.***

Specifically, we performed experiments with Apache Crail [5, 60] and DARE [49]. Crail is a bandwidth-hungry distributed data storage system that utilizes RDMA. In contrast, DARE is a latency-sensitive system that provides high-performance replicated state machines through the use of a strongly consistent RDMA-based key-value store.

In these experiments, we deployed DARE in a cluster of 4 nodes with 56 Gbps Mellanox ConnectX-3 Pro NIC on InfiniBand with 64GB memory. Crail is deployed in the same cluster with one node running the namenode and one other node running the datanode.

To evaluate the performance of Crail, we launch 8 parallel writes (each to a different file) in Crail's data storage with the chunk size of the data transfer configured to be 1MB, and we measure the application-level throughput reported by Crail. To evaluate the performance of DARE, one DARE client running on the same server as the namenode of Crail issues PUT and GET operations (each PUT is followed by a GET) to the DARE server on the other 3 nodes with a sweep of message sizes from 8 byte to 1024 bytes, and we measure the application-level latency reported by DARE.

Figure 6 plots the latency of DARE's queries with and without the presence of Crail. In this experiment, we observe a 4.6× increase in DARE's tail latency. Additionally, regardless of whether it is competing with DARE, Crail's total write throughput stays at 51.1 Gbps.

Besides DARE, highly-optimized RDMA-based RPC system such as FaSST [31] and eRPC [32] also suffer from isolation anomalies caused by unmanaged resource contention on RNICs. In fact, when background bandwidth-heavy traffic is present, FaSST's throughput experiences a 74% drop (Figure 32) and eRPC's tail latency increases by 40× (Figure 33). More details can be found in Appendix C.2.

3.3 Congestion Control is not Sufficient

To demonstrate that DCQCN [64] and PFC are not sufficient to solve these anomalies, we performed the benchmarks again with PFC enabled at both the NICs and switch ports, DCQCN [64] enabled at the NICs, and ECN markings enabled on a Dell 10 Gbps Ethernet switch (S4048-ON). In these experiments, latency- and throughput-sensitive applications still suffer unpredictably (Section 6.3 has detailed results). This is because DCQCN focuses on fabric-level isolation whereas

the observed anomalies happen at the end host due to RNIC resource contention (§2.2).

4 Justitia

Justitia enables multi-tenancy in hardware-based KBN, with a specific focus on enabling two performance-related policies: (1) fair/weighted resource sharing, or (2) predictable latencies while maximizing utilization, or a mix of the two. Note that we restrict our focus on a *cooperative* datacenter environment in this paper and defer strategyproofness [20, 21, 50] to mitigate adversarial/malicious behavior to future work.

Granularity of Control: We define a flow to be a stream of RDMA messages between two RDMA QPs. Justitia can be configured to work either at the flow granularity or at the application granularity by considering all flows between two applications as a whole.[3] In this paper, by default, we set Justitia's granularity of control to be at the application level to focus on application-level performance.

4.1 Key Design Ideas

Justitia resolves the unique challenges of enabling multi-tenancy in hardware KBN with five key design ideas.

- *Tenant-/application-level connection management:* To prevent tenants from hogging RNIC resources by issuing arbitrarily large messages or creating a large number of active QPs at no cost, Justitia provides a tenant-level connection management scheme by adding a shim layer between tenant applications and the RNIC. Tenant operations are handled by Justitia before arriving at the RNIC.
- *Sender-based proactive resource mediation:* Justitia proactively controls RNIC resource utilization at the sender side. This is based on the observation that the sender of an RDMA operation – that decides when an operation gets initiated, how large the message is, and in which direction the message flows – has active control over every aspect of the transmission while the other side of the connection remains passive. Such sender-based control can react before the RNIC takes over and maintain isolation by directly controlling RNIC resources.
- *Dynamic receiver-side updates:* Pure sender-based approaches can sometimes lead to spurious resource allocation when multiple senders coexist but are unaware of each other. Justitia leverage receiver-side updates to provide information (e.g., the arrival or departure of an application) back to the senders to react correctly when a change in the setting happens.
- *Passive latency monitoring:* Instead of actively measuring each application's latency, which can introduce high overhead, Justitia uses passive latency monitoring by issuing *reference flows* to detect RNIC resource contention.

[3] Each granularity has its pros and cons when it comes to performance isolation, without any conclusive answer on the right one [46].

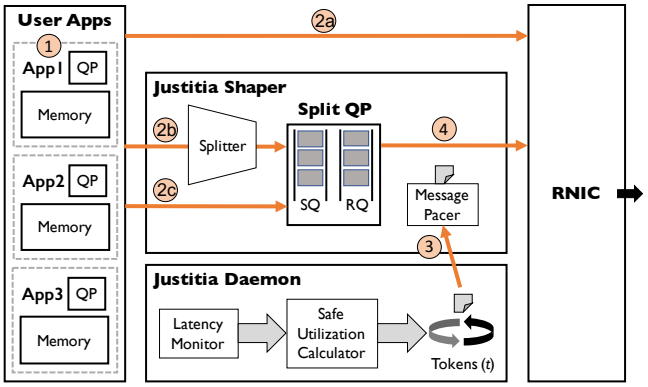


Figure 7: ***Justitia architecture. Bandwidth- and throughput-sensitive applications are shaped by tokens generated at a regular interval by Justitia. Latency-sensitive ones are not paced at all.***

- *Message-level shaping with splitting:* Justitia performs shaping at the message level to suit RDMA's message-oriented transport layer. At the message level, it is easy to apply specific strategies to control how messages enter the RNIC based on their sizes and the resource they consume. Large messages are split into roughly equal-sized sub-messages or chunks to (i) avoid a single message requesting too many RNIC resources; (ii) facilitate network sharing policies such as fair/weighted bandwidth share; and (iii) mitigate HOL Blocking for latency-sensitive applications.

4.2 System Overview

Figure 7 presents a high-level system overview of Justitia handling an RDMA WRITE operation (to compare with Figure 2). Each machine has a Justitia daemon that performs latency monitoring and proactive rate management, and applications create QPs using the existing API to perform RDMA communication. Justitia relies on applications to optionally identify their application type. By default, they are treated as bandwidth-sensitive. VMs, containers, bare-metal applications, and SR-IOV are all compatible with the design of Justitia.

As before, the user application starts an RDMA WRITE operation by ① posting a WQE into the Send Queue. Latency-sensitive applications will ②a bypass Justitia and directly interact with the RNIC as shown in Figure 2. The other two types of applications will enter Justitia's shaper. The Splitter will ②b split the big message from a bandwidth-sensitive applications equally into sub-messages or ②c do nothing given a small message from a throughput-sensitive application. We introduce *Split Connection* – and corresponding split queue pair (Split QP) – to handle the messages passed through the Splitter. Before sending out the message, it ③ asks the daemon to fetch a token from Justitia, which is generated at a rate to maximize RNIC resource utilization consumed by resource-hungry applications. Once the token is fetched, the Split QP ④ posts a WQE for the sub-message into its SQ and rings the door bell to notify the RNIC. The RNIC then grabs

the WQE from Split QP, issue a DMA read for the actual data in application's memory region, and sends the message to the remote side (arrows not shown in the figure). Steps ③ and ④ repeat until all messages in the Split QP have been processed. The implementation details of Split QP is in Section 5.1.

The Justitia daemon in Figure 7 is a background process that performs latency monitoring and proactive rate management to maximize RNIC resource utilization when latency target is met.

4.3 Justitia Daemon

Justitia daemon performs two major tasks: (i) proactively manages rate of all bandwidth- and throughput-sensitive applications using the hose model [17]; (ii) ensures predictable performance for latency-sensitive applications while maximizing RNIC resource usage.

4.3.1 Minimum Guaranteed Rate

Justitia enforces rate based on the classic hose model [17], and always maintains a minimum guaranteed rate R_{min}:

$$R_{min} = \frac{\sum w_B^i + \sum w_T^i}{\sum w_B^i + \sum w_T^i + \sum w_L^i} \times MaxRate$$

where w_X^i represents the weight of application i of type X (i.e., bandwidth-, throughput-, or latency-sensitive), and *MaxRate* represents the maximum RNIC bandwidth or maximum RNIC throughput (both are pre-determined on a per-RNIC basis) depending on the type of the application. The idea of R_{min} is to recognize the existence of latency-sensitive applications, and provide isolation for them by *taking out their share from the RNIC resources which otherwise they cannot acquire by themselves*. In the absence of latency-sensitive applications (i.e., $\sum w_L^i = 0$), R_{min} is equivalent to *MaxRate*, and all the resource-hungry applications share the entire RNIC resources. If all applications have equal weights, and there exist B bandwidth-, T throughput-, and L latency-sensitive applications, R_{min} can be simplified as $\frac{B+T}{B+T+L} \times MaxRate$.

In the presence of a large number of latency-sensitive applications, R_{min} could be really small, essentially removing RNIC resource guarantee. To accommodate such cases, one can fix $L = 1$ no matter how many latency-sensitive applications join the system since they do not consume much of RNIC's resources. We find this setting works well in practice (§6.4) and make it the default option for Justitia.

With R_{min} provided, Justitia then *maximizes RNIC's safe resource utilization* (which we denote *SafeUtil*) until the performance of latency-sensitive applications crosses the target tail latency ($Target_{99}$).

4.3.2 Latency Monitoring via Reference Flows

Justitia does not interrupt or interact with latency-sensitive applications because (i) they cannot saturate either of the two RNIC resources, and (ii) interrupting them fails to preserve RDMA's ultra-low latency.

Pseudocode 1 Maximize *SafeUtil*

```
1:  procedure OnLatencyFlowUpdate(L, Estimated99)
2:    if L = 0 then                ▷ Reset if no latency-sensitive applications
3:      SafeUtil = MaxRate
4:    else
5:      if Estimated99 > Target99 then
6:        SafeUtil = max(SafeUtil/2, Rmin)
7:      else
8:        SafeUtil = SafeUtil + 1
9:      end if
10:   end if
11:   τ = Token_Bytes / SafeUtil
12: end procedure
```

Instead, whenever there exists one or more latency-sensitive applications to particular receiving machine, Justitia maintains a *reference flow* to that machine which keeps sending 10B messages to the same receiver as the latency-sensitive applications in periodic intervals (by default, *RefPeriod* = 20 μs) to estimate the 99th percentile ($Estimated_{99}$) latency for small messages. By monitoring its own reference flow, Justitia does not need to wait on latency-sensitive applications to send a large enough number of sample messages for accurate tail latency estimation. It does not add additional delay by directly probing those applications either.

Given the stream of measurements, Justitia maintains a sliding window of the most recent *RefCount* (=10000) measurements for a reference flow estimate its tail latency.

4.3.3 Maximizing *SafeUtil*

Using the selected latency measurement from the reference flow(s), Justitia maximizes *SafeUtil* based on the algorithm shown in Pseudocode 1. To continuously update *SafeUtil*, Justitia uses a simple AIMD scheme that reacts to $Estimated_{99}$ every *RefPeriod* interval as follows. If the estimation is above $Target_{99}$, Justitia decreases *SafeUtil* by half; *SafeUtil* is guaranteed to be at least R_{min}. If the estimation is below $Target_{99}$, Justitia slowly increases *SafeUtil*. Because *SafeUtil* ranges between R_{min} to the total RNIC resources and latency-sensitive applications are highly sensitive to too high a utilization level, our conservative AIMD scheme, which drops utilization quickly to meet $Target_{99}$, works well in practice.

To determine the value of $Target_{99}$, we constructs a latency oracle that performs pair-wise latency measurement by issuing reference flows across all the nodes in the cluster when there is no other background. Microsoft applies a similar approach in [24], which is shown to work well in estimating steady-state latency in the cluster. We adopt this approach to give a good estimate of the latency target under well-isolated scenarios.

4.3.4 Token Generation And Distribution

Justitia uses multi-resource tokens to enforce *SafeUtil* among the B bandwidth- and T throughput-sensitive applications in a fair or weighted-fair manner. Each token represents a fixed

amount of bytes ($Token_{Bytes}$) and a fixed number of messages ($Token_{Ops}$). In other words, the size of $Token_{Bytes}$ determines the chunk size a message from bandwidth-sensitive application is split into. A token is generated every τ interval, where the value of τ depends on *SafeUtil* as well as on the size of each token. For example, given 48 Gbps application-level bandwidth and 30 Million operations/sec on a 56 Gbps RNIC, if $Token_{Bytes}$ is set to 1MB, then we set $Token_{Ops}$ =5000 ops and τ =167 μs.

Justitia daemon continuously generates one token every τ interval and distributes it among the active resource-hungry applications in a round-robin fashion based on application weights w_X^i. When $w_X^i = 1$ for all applications, Justitia enforces traditional max-min fairness; otherwise, it enforces weighted fairness. Each application independently enforces its rate using one of the shapers described below.

4.4 Justitia Shapers

Justitia shapers – implemented in the RDMA driver – enforce utilization limits provided by the Justitia daemon-calculated tokens. There are two shapers in Justitia: one for bandwidth- and another for throughput-sensitive applications.

Split Connection Justitia introduces the concept of a Split Connection to provide an interface to coordinate between tenant applications and the RNIC. It consists of a message splitter and custom *Split QPs* (§5.1) to initiate RDMA operations for tenants. Each application's Split Connection cooperate with Justitia daemon to pace split messages transparently.

Shaping Bandwidth-Sensitive Applications. This involves two steps: *splitting* and *pacing*. For any bandwidth-sensitive application, Justitia *transparently* divides any message larger than $Token_{Bytes}$ into $Token_{Bytes}$-sized chunks to ensure that the RNIC only sees roughly equal-sized messages. Splitting messages for diverse RDMA verbs – e.g., one-sided vs. two-sided – requires careful designing (§5.1).

Given chunk(s) to send, the pacer requests for token(s) from the Justitia daemon by marking itself as an active application. Upon receiving a token, it transfers chunk(s) until that token is exhausted and repeats until there is nothing left to send. The application is notified of the completion of a message only after all of its split messages have been transferred.

Batch Pacing for Throughput-Sensitive Applications. These applications typically deal with (batches of) small messages. Although there is no need for message splitting, pacing individual small messages requires the daemon to generate and distribute a large number of tokens, which can be CPU-intensive. Moreover, for messages as small as 16B, such fine-grained pacing cannot preserve RDMA's high message rates.

To address this, Justitia performs *batch pacing* enabled by Justitia's multi-resource token. Each token grants an application a fixed batch size ($Token_{Ops}$) that it can send together before receiving the next token. Batch pacing on throughput-sensitive applications removes the bottleneck on token generation and distribution; it also relieves daemon CPU cost with a unified token bucket.

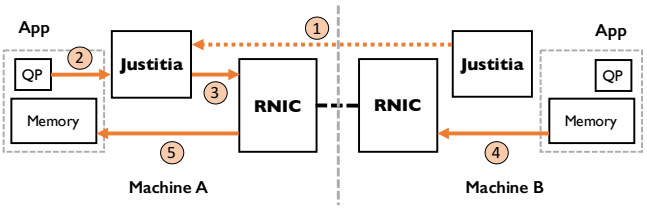


Figure 8: ***How Justitia handles READs via remote control.***

Mitigating Head-of-Line Blocking. One of the foremost goals of Justitia is to mitigate HOL blocking caused by the bandwidth-sensitive applications to provide predictable latencies. To achieve this goal, we need to split messages into smaller chunks and pace them at a certain rate (enforcing *SafeUtil*) with enough spacing between them to minimize the blocking. However, this simple approach creates a dilemma. On the one hand, too large a chunk may not resolve HOL Blocking. On the other hand, too small a chunk may not be able to reach *SafeUtil*. It also leads to increased CPU overhead from using a spin loop to fetch tokens generated in a very short period in which context switches are not affordable. This is a manifestation of the classic performance isolation-utilization tradeoff. We discuss how to pick the chunk size in Section 5.2.

4.5 Dynamic Receiver-Side Updates

Justitia relies on receiver-side updates to coordinate among multiple senders to avoid spurious allocation of RNIC resources. The benefits of this design is three-fold: (i) it coordinates with multiple senders to provide the correct resource allocation; (ii) it keeps track of RDMA READ issued which can collide with applications issuing RDMA WRITE in the opposite direction; (iii) it mitigates receiver-side engine congestion by rate-limiting senders with the correct fan-in information.

The updates are communicated among Justitia Daemons only when a change in the application state happened to a certain receiver (i.e., an arrival or an exit of an application) is detected. Two-sided operations, SEND and RECV, are selected in such case so that the daemon gets notified when an update arrives. Once a change is detected by a sender, it informs the receiver, which then broadcasts the change back to all the senders it connects to so that they can update the correct R_{min}. In such case, R_{min} considers remote resource-hungry application count as part of the total share. If the local daemon has not issued a reference flow and a remote latency-sensitive applications launches to the receiver, the daemon will start a new reference flow to start latency monitoring.

Handling READs RDMA specification allows remote machines to read from a local machine using the RDMA READ verb. RDMA READ operations issued by machine *A* to read data from machine *B* compete with all sending operations (e.g., RDMA WRITE) from machine *B*. Consequently, Justitia must handles remote READs as well.

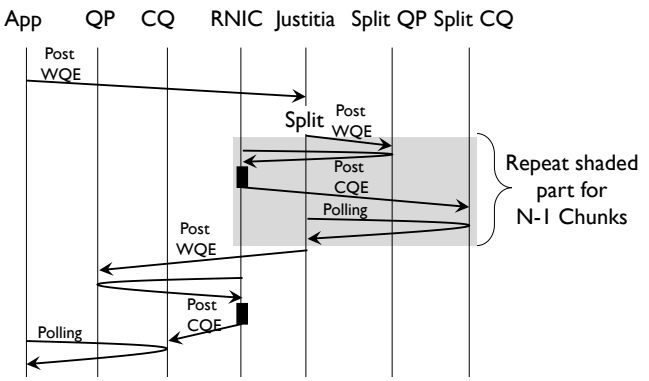


Figure 9: ***High-level overview of transparent message splitting in Justitia for one-sided verbs using Split QP. Times are not drawn to scale. Two-sided verbs involve extra bookkeeping.***

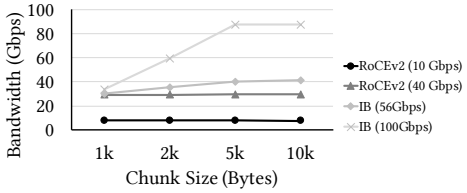


Figure 10: ***Maximum achievable bandwidth vs. chunk sizes.***

In such a case, the receiver of the READ operation, machine *B*, sends the updated guaranteed utilization R_{min}, with the updated count of senders including remote READ applications) as shown in ① in Figure 8. After *A* receives that utilization, it operates RDMA READ by interact with Justitia normally via ② → ⑤ and enforces the updated rate.

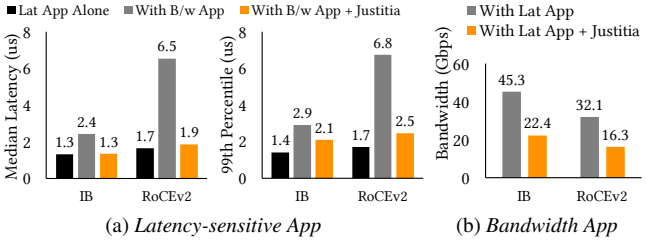
5 Implementation

We have implemented the Justitia daemon as a user-space process in 3,100 lines of C, and the shapers are implemented inside individual RDMA drivers with 5,200 lines of C code. Our current implementation focuses on container/bare-metal applications. Justitia code is available at https://github.com/SymbioticLab/Justitia.

5.1 Transparently Splitting RDMA Messages

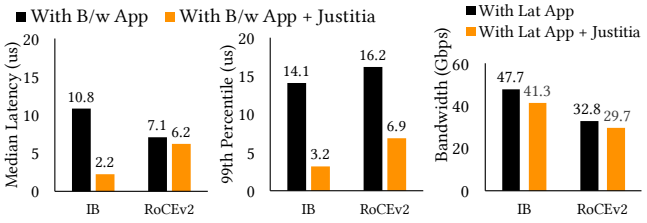
Justitia splitter transparently divides large messages of bandwidth-sensitive applications into smaller chunks for pacing. Our splitter uses a custom QP called a *Split QP* to handle message splitting, which is created when the original QP of a bandwidth-sensitive flow is created. A corresponding *Split CQ* is used to handle completion notifications. A custom completion channel is used to poll those notifications in an event-triggered fashion to preserve low CPU overhead.

To handle one-sided RDMA operations, when detecting a message larger than $Token_{Bytes}$, we divide the original message into chunks and only post the last chunk to the application's QP (Figure 9). The rest of the chunks are posted to the Split QP. Split QP ensures all chunks have been successfully transferred before the last chunk handled by the application's QP. The two-sided RDMA operations such as SEND are handled in a similar way, with additional flow control messages for the chunk size change and receive requests to be pre-posted at the receiver side.



(a) *Latency-sensitive App* (b) *Bandwidth App*

Figure 11: ***Performance isolation of a latency-sensitive application running against a bandwidth-sensitive one.***



(a) *Latency-sensitive App* (b) *Bandwidth App*

Figure 12: ***Latency of a latency-sensitive application running against a bandwidth-sensitive application with 4 QPs with a relaxed latency target (10 µs).***

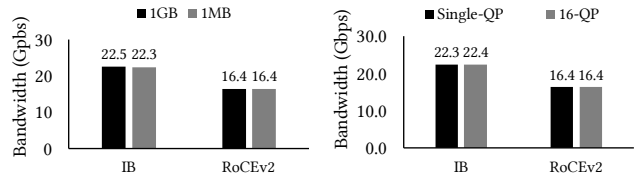
5.2 Determining Token Size for Bandwidth Target

One of the key steps in determining *SafeUtil* is deciding the size of each token. Because the RNIC can become throughput-bound for smaller messages instead of bandwidth-bound, we cannot use arbitrarily small messages to resolve HOL blocking. At the same time, given a utilization target, we want to use the smallest $Token_{Bytes}$ value to achieve that target to reduce HOL blocking while maximizing utilization.

Instead of dynamically determining it using another AIMD-like process, we observe that (i) this is an RNIC-specific characteristic and (ii) the number of RNIC types is small. With that in mind, we maintain a pre-populated dictionary to store the smallest token size that can saturate a given rate (to enforce *SafeUtil*) when sending in a paced batch for different latency targets; Justitia simply uses the mappings during runtime. When latency-sensitive applications are not present, a large token size (1MB) is used. Otherwise, Justitia looks up the token size in the dictionary based on the current *SafeUtil* value. This works well since the lower the *SafeUtil* is, the smaller the chunk size it requires to achieve such *SafeUtil*, and the better it helps mitigating HOL blocking. Based on our microbenchmarks (Figure 10), we pick 5KB as the chunk size when latency-sensitive applications are present.

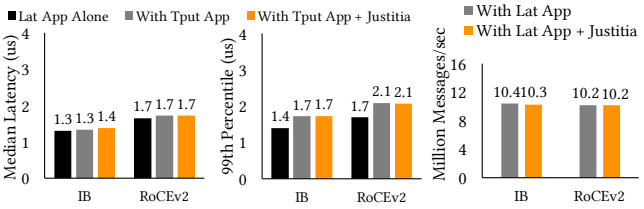
6 Evaluation

In this section, we evaluate Justitia's effectiveness in providing multi-tenancy support among latency-, throughput-, and bandwidth-sensitive applications on InfiniBand and RoCEv2. To measure latency, we perform 5 consecutive runs and present their median. We do not show error bars when they are too close to the median.



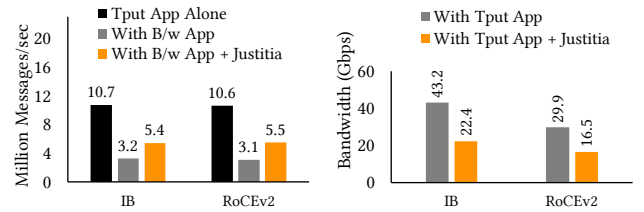
(a) *Different message sizes* (b) *Single-QP vs. 16-QP*

Figure 13: ***Fair bandwidth share of bandwidth-sensitive applications. (a) different message sizes. (b) different number of QPs.***



(a) *Latency-sensitive App* (b) *Throughput App*

Figure 14: ***Performance isolation of a latency-sensitive application running against a throughput-sensitive application.***



(a) *Throughput App* (b) *Bandwidth App*

Figure 15: ***Performance isolation of a throughput-sensitive application running against a bandwidth-sensitive application.***

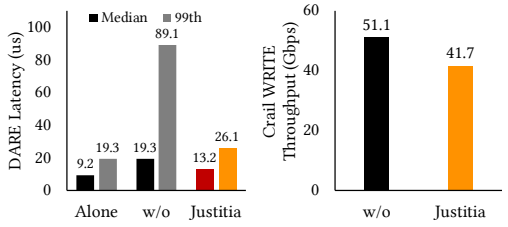
Our key findings can be summarized as follows:

- Justitia can effectively provide multi-tenancy support highlighted in Section 3 both in microbenchmarks and at the application-level (§6.1).
- Justitia scales well to a large number of applications and works for a variety of settings (§6.2); it complements DCQCN and hardware virtual lanes (§6.3).
- Justitia's benefits hold with many latency- and bandwidth-sensitive applications (§6.4), in incast scenarios (§6.5), and under unexpected network congestion (§6.6).

A detailed sensitivity analysis of Justitia parameters can be found in Appendix D.

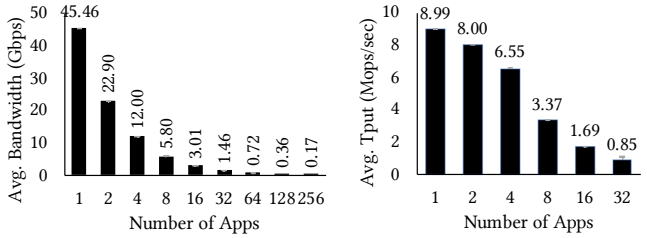
6.1 Providing Multi-Tenancy Support

We start by revisiting the scenarios from Section 3 to evaluate how Justitia enables sharing policies among different RDMA applications. We use the same setups as those in Section 3. Unless otherwise specified, we set $Target_{99}$ =2 μs on both InfiniBand and RoCEv2 for the latency-sensitive applications. Justitia works well in 100 Gbps networks too (Appendix C.1). Unless otherwise specified, R_{min} with all applications sharing the same weights is enforced as a default policy.



(a) *DARE Latency* (b) *CRAIL Throughput*

Figure 16: ***[InfiniBand] Performance isolation of DARE running against Crail.***



(a) *Bandwidth-sensitive* (b) *Throughput-sensitive*

Figure 17: ***[InfiniBand] Justitia scales to a large number of applications and still provides equal share. The error bars represent the minimum and the maximum values across all the applications.***

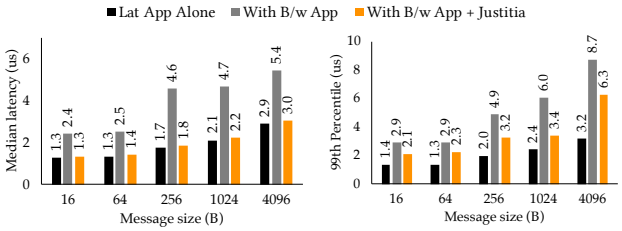


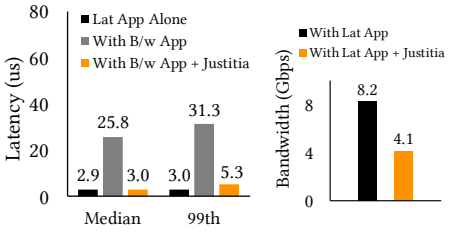
Figure 18: ***[InfiniBand] Latency-sensitive applications with different message sizes competing against a bandwidth-sensitive app.***

Predictable Latency Latency-sensitive applications are affected the most when they compete with a bandwidth-sensitive application. In the presence of Justitia, both median and tail latencies improve significantly in both InfiniBand and RoCEv2 (Figure 11a). Due to the enforcement of R_{min}, the bandwidth-sensitive application is receiving half of the capacity (Figure 11b).

Next we evaluate how Justitia performs when the latency target is set to a relaxed value ($Target_{99}$ =10 μs) that can be easily met (Figure 12). For a slightly high $Target_{99}$, Justitia maximizes utilization, illustrating that splitting and pacing are indeed beneficial.

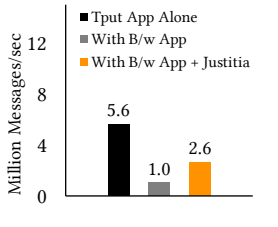
Fair Bandwidth and Throughput Sharing Justitia ensures that bandwidth-sensitive applications receive equal shares regardless of their message sizes and number of QPs in use (Figure 13) with small bandwidth overhead (less than 6% on InfiniBand and 2% on RoCEv2). The overhead becomes negligible when applying Justitia to throughput- or latency-sensitive applications (Figure 14).

Justitia's benefits extends to the bandwidth- vs through-sensitive application scenario as well. In this case, it ensures

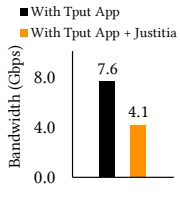


(a) *Latency-sensitive App* (b) *Bandwidth App*

Figure 19: ***[DCQCN] Latency-sensitive application against a bandwidth-sensitive one.***

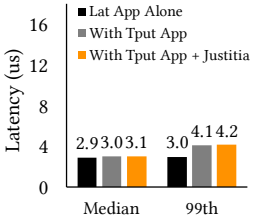


(a) *Tput App*

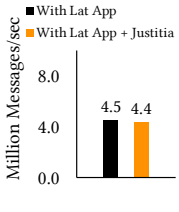


(b) *Bandwidth App*

Figure 20: ***[DCQCN] Throughput-sensitive app against a bandwidth-sensitive one.***



(a) *Latency-sensitive App*



(b) *Tput App*

Figure 21: ***[DCQCN] Latency-sensitive application against a throughput-sensitive one.***

that both receive roughly half of their resources. Figure 15 illustrates this behavior. In both InfiniBand and RoCEv2, the throughput-sensitive application is able to achieve half of its original message rate of itself running alone (Figure 15a). The bandwidth-sensitive application, on the other hand, is limited to half its original bandwidth as expected (Figure 15b).

Justitia and Real-World RDMA Applications To demonstrate that Justitia can isolate highly optimized real-world applications, we performed experiments with DARE and Crail. Thanks to Justitia's high transparency, we did not need to make any source code changes in Crail (given it is bandwidth-sensitive by default), and we only changed DARE by marking it as latency-sensitive.

From these experiments, we find that Justitia improves isolation for latency-sensitive applications while also preserving high bandwidth of the background storage application. Figure 16 plots the performance of DARE and Crail after applying Justitia with the same setting as in Section 3.2. We observe that, with Justitia, DARE achieves performance that is close to running in isolation even when running alongside Crail, and Justitia improves DARE's tail latency performance by 3.4× when compared to the baseline scenario while Crail also achieves 81% of its original throughput performance. This is close to the expected throughput of $\frac{8}{9}$ of Crail's original throughput since in this experiment Justitia treats the 8 parallel writes on top of Crail as separate applications.

Justitia improves performance isolation of FaSST by 2.5× in throughput and eRPC by 32.2× in tail latency. More details can be found in Appendix C.2.

6.2 Justitia Deep Dive

Scalability and Rate Conformance Figure 17a shows that as the number of bandwidth-sensitive applications increases, all applications receive the same amount of bandwidth using Justitia with total bandwidth close to the line rate. Justitia also ensures that all throughput-sensitive application send roughly equal number of messages (Figure 17b).

CPU and Memory Consumption Justitia daemon uses one dedicated CPU core per node to generate and distribute tokens. Its memory footprint is not significant.

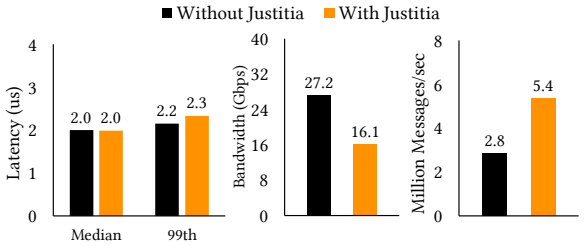


Figure 22: ***[RoCEv2] A bandwidth-, throughput-, and latency-sensitive application running on two hardware priority queues at the NIC. The latency-sensitive application uses one queue, while the other two share the other queue.***

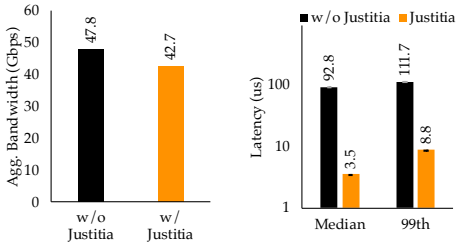
Varying Message Sizes Justitia can provide isolation at a wide range of message sizes for latency-sensitive applications (Figure 18). The bandwidth-sensitive application receives half the bandwidth in all cases.

6.3 Justitia + X

Justitia + DCQCN The anomalies we discover in this paper does not stem from the network congestion, but rather happens at the end hosts. We found that DCQCN falls short for latency- and throughput-sensitive applications (Figures 19, 20, 21). Justitia can complement DCQCN and improve latencies by up to 8.6× and throughput by 2.6×.

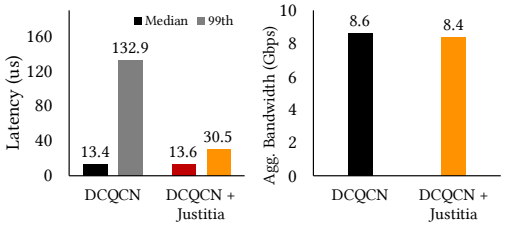
Justitia + Hardware Virtual Lanes Although RDMA standards support up to 15 virtual lanes [7] for separating traffic classes, they only map to very few hardware shapers and/or priority queues (2 queues in our RoCE NIC) that are rarely sufficient in shared environments [3, 37]. Besides, the hardware rate limiters in the RNIC are slow when setting new rates (2 milliseconds in our setup), making it hard to use with real dynamic arrangement. Moreover, it is desirable to achieve isolation *within each priority queue*, as those hardware resources are often used to provide different levels of quality of service, within which many applications reside.

In this experiment, we show how limited number of hardware queues are insufficient to provide isolation and how Justitia can help in this scenario. we run three applications, one each for each of the three types (Figure 22). Although the latency-sensitive application remains isolated in its own class, the bandwidth- and throughput-sensitive applications compete in the same class. As a result, the latter observes throughput loss (similar to Figure 15). Justitia can effectively provide performance isolation between bandwidth- and throughput-



(a) *Bandwidth Apps* (b) *Latency-Sensitive Apps*

Figure 23: ***[InfiniBand] Justitia isolating 8 latency-sensitive applications from 8 bandwidth-sensitive ones. Note that 8/9th of the bandwidth share is guaranteed since Justitia counts all latency-sensitive apps as one by default (§4.3.3).***



(a) *Latency* (b) *Bandwidth*

Figure 24: ***[DCQCN] Incast experiment with 33 senders and a single receiver. 32 senders launch bandwidth-sensitive applications, the other sender launches a latency-sensitive application.***

sensitive applications in the shared queue.

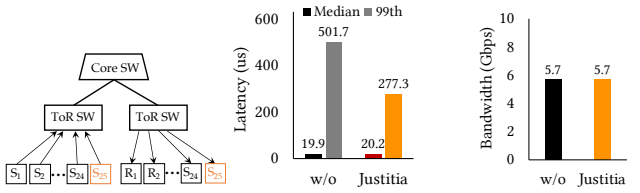
6.4 Isolating among More Competitors

We focus on Justitia's effectiveness in isolating many applications with different requirements and performance characteristics. Specifically, we consider 8 bandwidth-sensitive applications – 2 each with message sizes: 1MB, 10MB, 100MB, and 1GB, and 8 latency-sensitive applications. We measure the latency and bandwidth when all the applications are active in Figure 23. $Target_{99}$ is set to 2μs and 20 million samples are collected for latency measurements.

Without Justitia, latency-sensitive applications suffer large performance hits: individually each application had median and 99th percentile latencies of 1.3 and 1.4 μs (Figures 3a and 3b). With bandwidth-sensitive applications, they worsen by 71.4× and 79.8×. Justitia improves median and tail latencies of latency-sensitive applications by 26.5× and 12.7× while guaranteeing R_{min} among all the applications.

6.5 Handling Incast with Receiver-Side Updates

So far, we have focused on host-side RNIC contentions where the network fabric is not a bottleneck. We now evaluate how Justitia leverages receiver-side updates to handle receiver-side incast in both RoCEv2 with DCQCN and InfiniBand with its native credit-based flow control. In this experiment, 33 senders are used with the first 32 continuously launch a bandwidth-sensitive application sending 1MB messages to a single receiver. Simultaneously, the last sender launches a latency-sensitive application with messages sent to the same



(a) *Topology* (b) *Latency* (c) *Bandwidth*

Figure 25: ***[DCQCN] Justitia's performance when Inter-ToR links are congested. Justitia achieves the same bandwidth performance because the total amount of bandwidth share on S_{25} is smaller than SafeUtil due to other traffic flowing in the fabric.***

receiver. As described in Section 4.5, Justitia daemon at the receiver sends updates to all the senders whenever a sender application starts or exits, resulting in $\frac{1}{32}$-th of line rate guaranteed at each of the first 32 senders.

Figure 24 plots the results of this experiment, which show that Justitia still reduces tail latency even after the impact of fabric-level congestion on the reference flow latency measurements. Since the monitored latency misses the target, all the bandwidth-sensitive applications send at the minimum guaranteed rate. However, Justitia still achieves high aggregate bandwidth because this is greater than the fair share. This shows that Justitia complements congestion control and further improves the performance of latency-sensitive applications by mitigating receiver-side RNIC congestion.

We have also included a discussion on frequently asked questions regarding reference flows' impact in large-scale incast scenarios in Appendix E.4.

6.6 Justitia with Unexpected Network Congestion

When there is congestion inside the network, all traffic flowing through the network will experience increased latency, including the packets generated by Justitia as latency signals. Because today's switches and NICs do not report their individual contributions to end-to-end latency, Justitia cannot tell them apart. However, in practice, this is not a problem because the same response is appropriate in both scenarios.

To evaluate how Justitia performs under such cases, we performed experiments utilizing two interconnected ToR switches on CloudLab [11]. There are servers attached to each ToR switch, and every server has a line rate of 10Gbps. The experiment topology is shown in Figure 25a. In this topology, there is a third core switch that connects to each of the ToR switches with a link with a capacity of 160 Gbps. In this experiment, we enable DCQCN at all the servers and ECN marking at the ToR switches in the cluster.

To create a congested ToR uplink, we launch 24 bandwidth-sensitive applications each issuing 1MB messages from 24 servers (S_1–S_{24}) under one rack to the other 24 servers (R_1–R_{24}) under another rack, and none of the servers run Justitia. At the same time, we issue 8 bandwidth-sensitive applications and 1 latency-sensitive application between a pair of servers (S_{25} and R_{25}) that is controlled by Justitia. Figure 25 shows the performance with and without Justitia applied. Even in

the case where fabric congestion is out of Justitia's control, we see that Justitia can still function correctly, and Justitia still provides additional performance isolation benefits when compared with just using congestion control (DCQCN).

7 Related Work

RDMA Sharing Recently, large-scale RDMA deployment over RoCEv2 have received wide attention [23, 41, 44, 45, 64]. However, the resulting RDMA congestion control algorithms [40, 41, 44, 64] primarily deal with Priority-based Flow Control (PFC) to provide fair sharing between bandwidth-sensitive applications inside the network. In contrast, Justitia focuses on RNIC isolation and complements them (§6.3).

Justitia is complementary to FreeFlow [35] as well. FreeFlow enables *untrusted* containers to securely preserve the performance benefits of RDMA. Because it does not change how verbs are sent to queue pairs, it can still suffer from the performance isolation problems Justitia addresses. Justitia can complement FreeFlow to provide performance isolation by implementing Justitia splitter in FreeFlow's network library and Justitia daemon in its virtual router.

SR-IOV [55] is a hardware-based I/O virtualization technique that allows multiple VMs to access the same PCIe device on the host machine. Justitia design does not interfere with SR-IOV and will still work on top of it. To provide multi-tenant fairness, Justitia can be modified to distribute credits among VMs via shared memory channel similar to [35].

LITE [62] also addresses resource sharing and isolation issues in RNICs. However, LITE does not perform well in the absence of hardware virtual lanes (Appendix C.4).

PicNIC [38] tries to provide performance isolation at the receiver-side engine congestion in software-based kernel-bypass networks, where it utilizes user-level packet processing instead of offloading packetization to an RNIC. Hence, PicNIC's CPU-based resource allocation and packet-level shaping cannot be applied to RDMA.

Swift [36] also considers receiver-side enginer congestion in software KBN by using a dedicated enging congestion window in the congestion algorithm. However, both Swift and PicNIC ignores sender-side congestion.

Offloading with SmartNICs Recent research in SmartNICs has focused on providing programmability and efficiency in hardware offloading [6, 19, 33, 39, 42]. However, on-NIC packet orchestration leads to tens of microsecond overhead [19, 57], making performance-related multi-tenancy support still an open problem.

NICA [18] provides isolation for FPGA-based SmartNICs by I/O channal virtualization and time-sharing of the Acceleration Functional Units. Justitia focus on normal RNICs and does not require hardware changes.

Link Sharing Max-min fairness [9, 15, 28, 54] is the well-established solution for link sharing that achieves both sharing incentive and high utilization, but it only considers bandwidth-sensitive applications. Latency-sensitive applications can rely on some form of prioritization for isolation [3, 25, 63].

Although DRFQ [20] deals with multiple resources, it considers cases where a packet sequentially accessed each resource, both link capacity and latency were significantly different than RDMA, and the end goal is to equalize utilization instead of performance isolation. Furthermore, implementing DRFQ required hardware changes.

Both Titan [58] and Loom [56] improve performance isolation on conventional NICs by programming on-NIC packet schedulers. However, this is not sufficient for RDMA performance isolation because it schedules only the outgoing link. Further, Justitia works on existing RNICs that are opaque and do not have programmable packet schedulers.

TAS [34] accelerates TCP stack by separating the TCP fast-path from OS kernel to handle packet processing and resource enforcement. However, TAS does not solve the type of isolation anomalies Justitia deals with. Justitia's design idea can be applied to improve isolation for TAS.

Datacenter Network Sharing With the advent of cloud computing, the focus on link sharing has expanded to network sharing between multiple tenants [4, 8, 10, 46, 50, 53]. Almost all of them – except for static allocation – deal with bandwidth isolation and ignore predicted latency on latency-sensitive applications.

Silo [29] deals with datacenter-scale challenges in providing latency and bandwidth guarantees with burst allowances on Ethernet networks. In contrast, we focus on isolation anomalies in multi-resource RNICs between latency-, bandwidth-, and throughput-sensitive applications.

8 Concluding Remarks

We have demonstrated that RDMA's hardware-based kernel bypass mechanism has resulted in lack of multi-tenancy support, which leads to performance isolation anomalies among bandwidth-, throughput-, and latency-sensitive RDMA applications across InfiniBand, RoCEv2, and iWARP and in 10, 40, 56, and 100 Gbps networks. We presented Justitia, which uses a combination of sender-based resource mediation with receiver-side updates, Split Connection with message-level shaping, and passive machine-level latency monitoring, together with a tail latency target as a single knob to provide network sharing policies for RDMA-enabled networks.

Acknowledgments

Special thanks go to the CloudLab and ConFlux teams for enabling most of the experiments and to RTCL for some early experiments on RoCEv2. We would also like to thank all the anonymous reviewers, our shepherd, Costin Raiciu, and SymbioticLab members for their insightful feedback. This work was supported in part by NSF grants CNS-1845853, CNS-1909067, and CNS-2104243, gifts from VMware and Google, and an equipment gift from Chelsio Communications.

References

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

[2] Marcos K. Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Stanko Novakovic, Arun Ramanathan, Pratap Subrahmanyam, Lalith Suresh, Kiran Tati, Rajesh Venkatasubramanian, , and Michael Wei. Remote regions: a simple abstraction for remote memor. In *ATC*, 2018.

[3] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick Mckeown, Balaji Prabhakar, and Scott Shenker. pFabric: Minimal near-optimal datacenter transport. In *SIGCOMM*, 2013.

[4] Sebastian Angel, Hitesh Ballani, Thomas Karagiannis, Greg O'Shea, and Eno Thereska. End-to-end performance isolation through virtual datacenters. In *OSDI*, 2014.

[5] Apache. Apache crail. http://crail.incubator.apache.org/., 2021.

[6] Mina Tahmasbi Arashloo, Alexey Lavrov, Manya Ghobadi, Jennifer Rexford, David Walker, and David Wentzlaff. Enabling programmable transport protocols in high-speed nics. In *NSDI*, 2020.

[7] Infiniband Trade Association. Infiniband architecture specification volume 1. https://cw.infinibandta.org/document/dl/7859, 2015.

[8] Hitesh Ballani, Paolo Costa, Thomas Karagiannis, and Ant Rowstron. Towards predictable datacenter networks. In *SIGCOMM*, 2011.

[9] J.C.R. Bennett and H. Zhang. WF^2Q: Worst-case fair weighted fair queueing. In *INFOCOM*, 1996.

[10] M. Chowdhury, Z. Liu, A. Ghodsi, and I. Stoica. HUG: Multi-resource fairness for correlated and elastic demands. In *NSDI*, 2016.

[11] Cloudlab. http://cloudlab.us/.

[12] RL Cruz. A calculus for network delay, Part I: Network elements in isolation. *IEEE Transactions on Information Theory*, 37(1):114–131, 1991.

[13] RL Cruz. A calculus for network delay, Part II: Network analysis. *IEEE Transactions on Information Theory*, 37(1):132–141, 1991.

[14] Michael Dalton, David Schultz, Jacob Adriaens, Ahsan Arefin, Anshuman Gupta, Brian Fahs, Dima Rubinstein, Enrique Cauich Zermeno, Erik Rubow, James Alexander Docauer, Jesse Alpert, Jing Ai, Jon Olson, Kevin DeCabooter, Marc de Kruijf, Nan Hua, Nathan Lewis, Nikhil Kasinadhuni, Riccardo Crepaldi, Srinivas Krishnan, Subbaiah Venkata, Yossi Richter, Uday Naik, , and Amin Vahdat. Andromeda: Performance, isolation, and velocity at scale in cloud network virtualization. In *NSDI*, 2018.

[15] A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair queueing algorithm. In *SIGCOMM*, 1989.

[16] Aleksandar Dragojevic, Dushyanth Narayanan, Orion Hodson, and Miguel Castro. FaRM: Fast remote memory. In *NSDI*, 2014.

[17] Nick G. Duffield, Pawan Goyal, Albert Greenberg, Partho Mishra, Kadangode K Ramakrishnan, and Jacobus E van der Merwe. A flexible model for resource management in virtual private networks. In *SIGCOMM*, 1999.

[18] Haggai Eran, Lior Zeno, Maroun Tork, Gabi Malka, and Mark Silberstein. NICA: An infrastructure for inline acceleration of network applications. In *USENIX ATC*, 2019.

[19] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, Harish Kumar Chandrappa, Somesh Chaturmohta, Matt Humphrey, Jack Lavier, Norman Lam, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A. Maltz, , and Albert Greenberg. Azure accelerated networking: Smartnics in the public cloud. In *NSDI*, 2018.

[20] Ali Ghodsi, Vyas Sekar, Matei Zaharia, and Ion Stoica. Multi-resource fair queueing for packet processing. 2012.

[21] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. Dominant resource fairness: Fair allocation of multiple resource types. In *NSDI*, 2011.

[22] J. Gu, Y. Lee, Y. Zhang, M. Chowdhury, and K. G. Shin. Efficient memory disaggregation with Infiniswap. In *NSDI*, 2017.

[23] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. RDMA over commodity Ethernet at scale. In *SIGCOMM*, 2016.

[24] Chuanxiong Guo, Lihua Yuan, Dong Xiang, Yingnong Dang, Ray Huang, Dave Maltz, Zhaoyi Liu, Vin Wang, Bin Pang, Hua Chen, Zhi-Wei Lin, and Varugis Kurien. Pingmesh: A large-scale system for data center network latency measurement and analysis. In *SIGCOMM*, 2015.

[25] Chi-Yao Hong, Matthew Caesar, and P. Brighten Godfrey. Finishing flows quickly with preemptive scheduling. In *SIGCOMM*, 2012.

[26] Cheng Huang, Huseyin Simitci, Yikang Xu, Aaron Ogus, Brad Calder, Parikshit Gopalan, Jin Li, and Sergey Yekhanin. Erasure coding in Windows Azure Storage. In *USENIX ATC*, 2012.

[27] Intel. HTB Home. http://luxik.cdi.cz/~devik/qos/htb/, 2003.

[28] Jeffrey M Jaffe. Bottleneck flow control. *IEEE Transactions on Communications*, 29(7):954–962, 1981.

[29] Keon Jang, Justine Sherry, Hitesh Ballani, and Toby Moncaster. Silo: Predictable message latency in the cloud. In *SIGCOMM*, 2015.

[30] Anuj Kalia, Michael Kaminsky, and David G Andersen. Using RDMA efficiently for key-value services. In *SIGCOMM*, 2014.

[31] Anuj Kalia, Michael Kaminsky, and David G Andersen. FaSST: fast, scalable and simple distributed transactions with two-sided (RDMA) datagram RPCs. In *OSDI*, 2016.

[32] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Datacenter rpcs can be general and fast. In *NSDI*, 2019.

[33] Antoine Kaufmann, Simon Peter, Naveen Kr. Sharma, Thomas Anderson, and Arvind Krishnamurthy. High performance packet processing with flexnic. In *ASPLOS*, 2016.

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

[35] Daehyeok Kim, Tianlong Yu, Hongqiang Harry Liu, Yibo Zhu, Jitu Padhye, Shachar Raindel, Chuanxiong Guo, Vyas Sekar, and Srinivasan Seshan. Freeflow: Software-based virtual RDMA networking for containerized clouds. In *NSDI*, 2019.

[36] Gautam Kumar, Nandita Dukkipati, Keon Jang, Hassan M. G Wassel, Xian Wu, Yaogong Montazeri, Behnam andand Wang, Kevin Springborn, Christopher Alfeld, Michael Ryan, David Wetherall, and Amin Vahdat. Swift: Delay is simple and effective for congestion control in the datacenter. In *SIGCOMM*, 2020.

[37] Gautam Kumar, Srikanth Kandula, Peter Bodik, and Ishai Menache. Virtualizing traffic shapers for practical resource allocation. In *HotCloud*, 2013.

[38] Praveen Kumar, Nandita Dukkipati, Nathan Lewis, Yi Cui, Yaogong Wang, Chonggang Li, Valas Valancius, Adriaens Jake, Steve Gribble, Nate Foster, and Amin Vahdat. PicNIC: Predictable virtualized nic. In *SIGCOMM*, 2019.

[39] Yanfang Le, Hyunseok Chang, Sarit Mukherjee, Limin Wang Aditya Akella, Michael M. Swift, and T.V. Lakshman. Uno: Unifying host and smart nic offload for flexible packet processing. In *SoCC*, 2017.

[40] Yanfang Le, Brent Stephens, Arjun Singhvi, Aditya Akella, and Michael M. Swift. RoGUE: RDMA over generic unconverged ethernet. In *SoCC*, 2018.

[41] Yuliang LI, Harry Hongqiang Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, and Minlan Yu. Hpcc: High precision congestion control. In *SIGCOMM*, 2019.

[42] Ming Liu, Tianyi Cui, Henry Schuh, Arvind Krishnamurthy, Simon Peter, and Karan Gupta. Offloading distributed applications onto smartnics using ipipe. In *SIGCOMM*, 2019.

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

[44] Radhika Mittal, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zats. TIMELY: RTT-based congestion control for the datacenter. In *SIGCOMM*, 2015.

[45] Radhika Mittal, Alexander Shpiner, Aurojit Panda, Eitan Zahavi, Arvind Krishnamurthy, Sylvia Ratnasamy, and Scott Shenker. Revisiting network support for RDMA. In *SIGCOMM*, 2018.

[46] Jeffrey C Mogul and Lucian Popa. What we talk about when we talk about cloud network performance. *SIGCOMM CCR*, 42(5):44–48, 2012.

[47] Jacob Nelson, Brandon Holt, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, and Mark Oskin. Latency-tolerant software distributed shared memory. In *USENIX ATC*, 2015.

[48] Jonathan Perry, Amy Ousterhout, Hari Balakrishnan, Devavrat Shah, and Hans Fugal. Fastpass: A centralized zero-queue datacenter network. 2014.

[49] Marius Poke and Torsten Hoefler. DARE: High-performance state machine replication on rdma networks. In *HPDC*, 2015.

[50] L. Popa, G. Kumar, M. Chowdhury, A. Krishnamurthy, S. Ratnasamy, and I. Stoica. FairCloud: Sharing the network in cloud computing. In *SIGCOMM*, 2012.

[51] Ahmed Saeed, Nandita Dukkipati, Vytautas Valancius, Carlo Contavalli, Amin Vahdat, et al. Carousel: Scalable traffic shaping at end hosts. In *SIGCOMM*, 2017.

[52] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. LegoOS: A disseminated, distributed os for hardware resource disaggregation. In *OSDI*, 2018.

[53] Alan Shieh, Srikanth Kandula, Albert Greenberg, and Changhoon Kim. Sharing the data center network. In *NSDI*, 2011.

[54] Madhavapeddi Shreedhar and George Varghese. Efficient fair queuing using deficit round-robin. *IEEE/ACM Transactions on Networking*, 4(3):375–385, 1996.

[55] SR-IOV. Single root i/o virtualization. http://pcisig.com/specifications/iov/single_root/., 2018.

[56] Brent Stephens, Aditya Akella, and Michael Swift. Loom: Flexible and efficient nic packet scheduling. In *NSDI*, 2017.

[57] Brent Stephens, Aditya Akella, and Michael Swift. Your programmable nic should be a programmable switch. In *HotNets*, 2018.

[58] Brent Stephens, Arjun Singhvi, Aditya Akella, and Michael Swift. Titan: Fair packet scheduling for commodity multiqueue nics. In *USENIX ATC*, 2017.

[59] I. Stoica, H. Zhang, and T.S.E. Ng. A hierarchical fair service curve algorithm for link-sharing, real-time and priority service. In *SIGCOMM*, 1997.

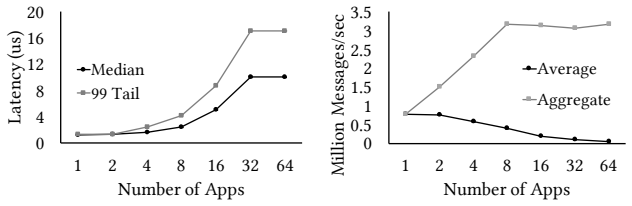
[60] Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, Ana Klimovic, Adrian Schuepbach, and Bernard Metzler. Unification of temporary storage in the nodekernel architecture. In *ATC*, 2019.

[61] Mellanox Technologies. Mellanox Perftest Package. https://community.mellanox.com/docs/DOC-2802, 2017.

[62] Shin-Yeh Tsai and Yiying Zhang. LITE kernel rdma support for datacenter applications. In *SOSP*, 2017.

[63] Christo Wilson, Hitesh Ballani, Thomas Karagiannis, and Ant Rowtron. Better never than late: Meeting deadlines in datacenter networks. In *SIGCOMM*, 2011.

[64] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. Congestion control for large-scale RDMA deployments. In *SIGCOMM*, 2015.



(a) *Latency* (b) *Throughput*

Figure 26: ***Latencies and throughputs of multiple latency-sensitive applications in InfiniBand. Error bars (almost invisible due to close proximity) represent the applications with the lowest and highest values.***

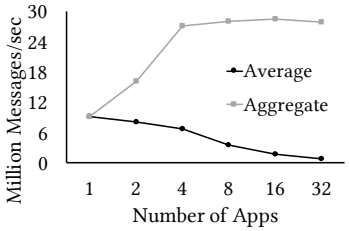
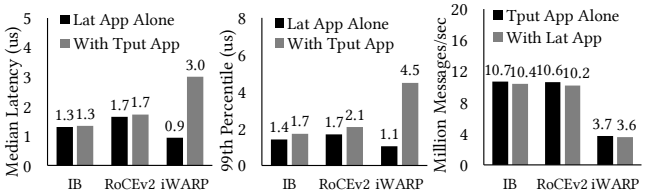


Figure 27: ***Throughputs of multiple throughput-sensitive applications in InfiniBand. Error bars (almost invisible due to close proximity) represent the applications with the lowest and highest values.***



(a) *Latency App (Med)* (b) *Latency App (99th)* (c) *Throughput App*

Figure 28: ***Performance anomalies of a latency-sensitive application running against a throughput-sensitive application.***

A Hardware Testbed Summary

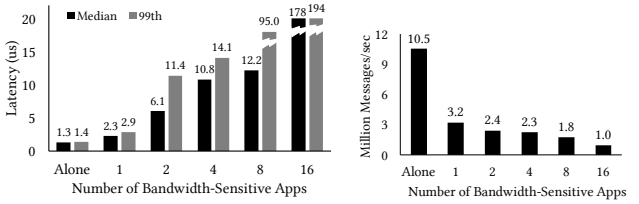
Table 1 summarizes the hardware we use for different RDMA protocols in our experiments.

B Characteristics of Latency- and Throughput-Sensitive Applications in the Absence of Bandwidth-Sensitive Ones

Multiple latency-sensitive applications can coexist without affecting each other (Figure 26). Although latencies increase, everyone suffers equally. All applications experience the same throughputs as well.

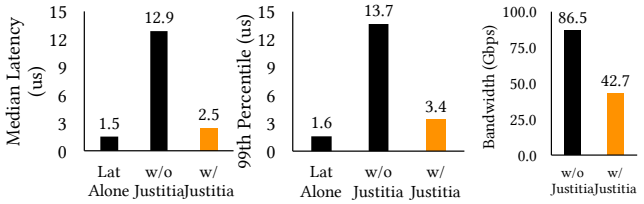
Similarly, multiple throughput-sensitive applications receive almost equal throughputs when competing with each other, as shown in Figure 27.

Finally, throughput-sensitive applications do not get affected by much when competing with latency-sensitive applications (Figure 28c). Nor do latency-sensitive applications experience noticeable latency degradations in the presence of throughput-sensitive applications except for iWARP (Figure 28a and Figure 28b).



(a) *Impact on Latency* (b) *Impact on Throughput*

Figure 29: ***Impact of increasing background bandwidth-sensitive applications (sending 1MB messages) in InfiniBand.***



(a) *Latency-sensitive App* (b) *Bandwidth App*

Figure 30: ***[100 Gbps InfiniBand] Performance isolation of a latency-sensitive application against a bandwidth-sensitive application.***

B.1 Adding More Competitors Exacerbates the Anomalies

The lack of protection for the latency-sensitive applications further exacerbates as more bandwidth-sensitive applications (or equivalently more QPs) are created. We increase the number of bandwidth-sensitive applications (each with a single QP) in our experiment to simulate more realistic datacenter applications. Although InfiniBand performs relatively well in the presence of a single background bandwidth-sensitive application (Figure 3), adding one more competitors incurs an additional drop of 2.65× and 3.79× in median and 99th percentile latencies (Figure 29a). With 16 or more bandwidth-sensitive applications, the latency-sensitive application can barely make any progress. We observed a similar trend in other RDMA technologies.

Similarly, a throughput-sensitive application loses 90% of its original throughput with 16 bandwidth-sensitive applications (Figure 29b).

Those anomalies illustrate RNIC's inability to handle multiple types of applications, which could stem from the limited number of queues inside the RNIC hardware, increasing Head-of-Line blocking of small messages.

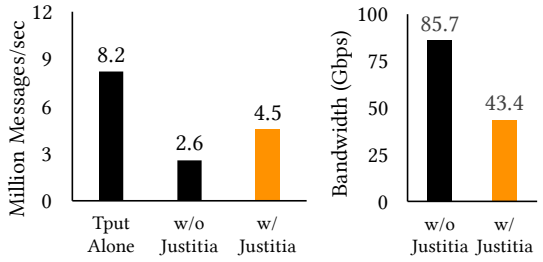
C Additional Evaluation Results

C.1 100 Gbps Results With/Without Justitia

Similar to the anomalies observed for 10, 40, and 56 Gbps networks (§3), Figure 30 and Figure 31 show that latency- and throughput-sensitive applications are not isolated from bandwidth-sensitive applications even in 100 Gbps networks. In these experiments, we use 5MB messages since 1MB messages are not large enough to saturate the 100 Gbps link. Justitia can effectively mitigate the challenges by enforcing performance isolation.

Protocol	NIC	Switch	NIC Capacity
InfiniBand	ConnectX-3 Pro	Mellanox SX6036G	56 Gbps
InfiniBand	ConnectX-4	Mellanox SB7770	100 Gbps
RoCEv2	ConnectX-4	Mellanox SX6018F	40 Gbps
RoCEv2 (DCQCN) (§6.3)	ConnectX-4 Lx	Dell S4048-ON	10 Gbps
iWARP	T62100-LP-CR	Mellanox SX6018F	40 Gbps
RoCEv2 (DCQCN) (§6.5 and §6.6)	ConnectX-3 Pro	HP Moonshot-45XGc	10 Gbps

Table 1: ***Testbed hardware specification.***



(a) *Throughput App* (b) *B/w App*

Figure 31: ***[100 Gbps InfiniBand] Performance isolation of a throughput-sensitive application against a bandwidth-sensitive application.***

C.2 Real RDMA-based Systems Require Isolation

Besides DARE, highly-optimized RDMA-based RPC systems also suffer from unmanaged RNIC resources. Here we pick two representative systems, FaSST [31] and eRPC [32], to illustrate why they require performance isolation and how Justitia effectively achieves it. To generate background traffic, we implemented a simple RDMA-based blob storage backend across 16 machines. Users read/write data to this storage using a PUT/GET interface via frontend servers. Objects larger than 1MB are divided into 1MB splits and distributed across the backend servers. This generates a stream of 1MB transfers, and the following RDMA-optimized systems have to compete with them in our experimental setup.

FaSST is an RDMA-based RPC system optimized for high message rate. We deploy FaSST in 2 nodes with message size of 32 bytes and a batch size of 8. We use 4 threads to saturate FaSST's message rate at 9.8 Mrps. In the presence of the storage application, FaSST's throughput experiences a 74% drop (Figure 32).

eRPC is an even more recent RPC system built on top of RDMA. We deploy eRPC in 2 nodes with message size of 32 bytes. We evaluate eRPC's latency and throughput using the microbenchmark provided by its authors. For the throughput experiment, we use 2 worker threads with a batch size of 8 on each node because 2 threads are enough to saturate the message rate in our 2-node setting. In the presence of the storage application, eRPC's throughput drops by 93% (Figure 33b), and its median and tail latencies increase by 67× and 40×, respectively (Figure 33a).

By applying Jusitita, FaSST's throughput improves by 2.5× (Figure 32). Justitia also improves eRPC's median (tail)

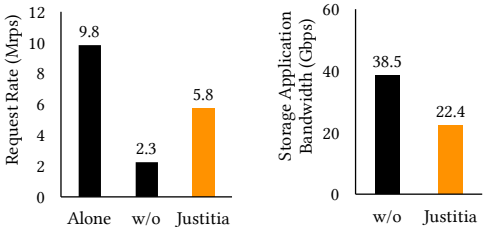
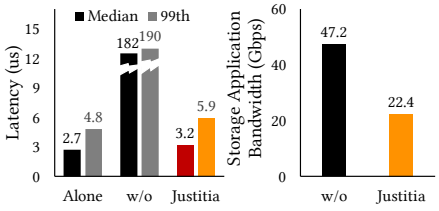
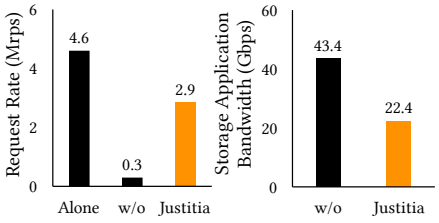


Figure 32: ***Performance isolation of FaSST running against a bandwidth-sensitive storage application.***



(a) *Latency*



(b) *Throughput*

Figure 33: ***Performance isolation of eRPC running against a bandwidth-sensitive storage application.***

latency improves by 56.9× (32.2×) and its throughput by 9.7× (Figure 33). Note that the throughput of the storage applications drops to half of the maximum throughput in both cases because we treat the background application as a whole (and thus with equal weights to all applications, the *SafeUtil* is $\frac{1}{2}$ of the line rate), which is different from how we treat the parallel writes in the case of Apache Crial.

C.3 Handling Remote READs

RDMA READ verbs can compete with WRITEs and SENDs issued from the opposite direction (§4.5) Figure 34 shows that Justitia can isolate latency-sensitive remote READs from local bandwidth-sensitive WRITEs and vice versa.

C.4 Justitia vs. LITE

LITE [62] is a software-based RDMA implementation that adds a local indirection layer for RDMA in the Linux kernel

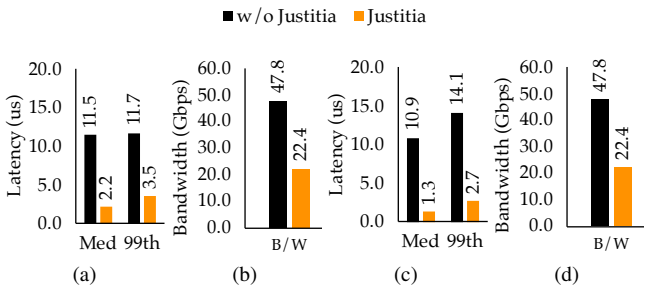
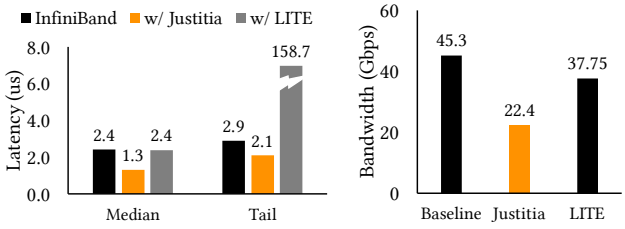


Figure 34: ***[InfiniBand] (a)–(b) Justitia isolating remote latency-sensitive READs from local bandwidth-sensitive WRITEs. (c)–(d) Justitia isolating local latency-sensitive WRITEs from remote bandwidth-sensitive READs.***



(a) *Latency-sensitive App* (b) *Bandwidth App*

Figure 35: ***[InfiniBand] Performance isolation of a latency-sensitive flow running against a 1MB background bandwidth-sensitive flow using Justitia and LITE.***

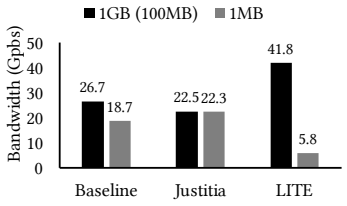


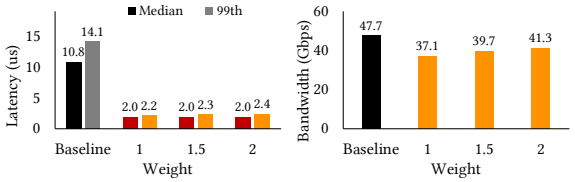
Figure 36: ***[InfiniBand] Bandwidth allocations of two bandwidth-sensitive applications using Justitia and LITE. LITE uses 100MB messages instead of 1GB due to its own limitation.***

to virtualize RDMA and enable resource sharing and performance isolation. It can use hardware virtual lanes and also includes a software-based prioritization scheme.

We found that, in the absence of hardware virtual lanes, LITE does not perform well in isolating latency-sensitive flow from the bandwidth-sensitive one (Figure 35) – 122× worse 99th percentile latency than Justitia. In terms of bandwidth-sensitive applications using different message sizes, LITE performs even worse than native InfiniBand (Figure 36). Justitia outperforms LITE's software-level prioritization by being cognizant of the tradeoff between performance isolation and high utilization.

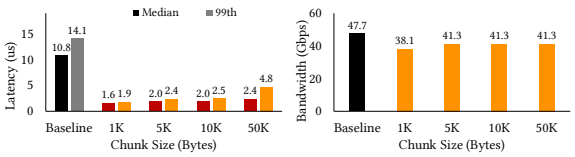
D Sensitivity Analysis

Setting Applications Weights To evaluate how assigning different application weights (§4.3.1) affects Justitia's performance, we launch 4 bandwidth-sensitive applications each sending 1MB message together with a latency-sensitive application, and we vary the weights of the bandwidth-sensitive applications. Figure 37 illustrates the impact of setting different application weights with latency target set to 2 μs. As



(a) *Latency* (b) *Bandwidth*

Figure 37: ***[InfiniBand] Sensitivity analysis of application weights.***



(a) *Latency* (b) *Bandwidth*

Figure 38: ***[InfiniBand] Sensitive analysis of chunk sizes.***

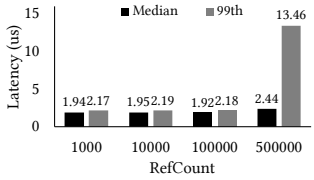


Figure 39: ***[InfiniBand] Sensitivity analysis of RefCount.***

the weight increases, the value of *SafeUtil* increases, and thus more aggregate bandwidth share is obtained for bandwidth-sensitive applications. Higher *SafeUtil* leads to worse latency isolation, but in these experiments the effect of weights on tail latency performance is not huge. In fact, we do not find much latency performance degradation as the weight increases, illustrating the effectiveness of Justitia mitigating head-of-line blocking via its splitting mechanism. The cluster operator can choose weights based on the priority of the applications in the cluster based on the Quality-of-Service inside the cluster (similar to deciding bandwidth resources in a shared environment), or based on how much each application pays to obtain the service. Additionally, if multi-tenant fairness is desired, one can achieve that by modifying how credits are allocated in Justitia on a per-tenant basis. Justitia supports allocating tokens at multiple granularities if needed, which can be per-tenant, per-application, or per group of connections within an application.

Setting Chunk Size When latency-sensitive applications are present, Justitia picks the smallest chunk size that still provides a wide range of bandwidth in case *SafeUtil* is high (Figure 10). Here we evaluate how setting the correct chunk size affects Justitia's performance. We use the same setting as the sensitivity analysis of application weights and set the weight to be 2. Figure 38 illustrates the experiment results with different chunk sizes. Although a smaller chunk size provides better latency isolation, it is not able to achieve *SafeUtil* and thus waste bandwidth resources. On the other hand, too big of a chunk size does not provide enough latency isolation.

Setting *RefCount* To evaluate how sensitive the value *RefCount* (§4.3.2) is, we design an experiment where initially we launch one latency-sensitive application to compete with a bandwidth-sensitive application. Once the experiment starts, we add three additional bandwidth-sensitive application, with a gap of 1 second between their arrival time. We measure the latency of latency-sensitive application after it completes 10 million messages. Figure 39 plots the results with different *RefCount* values. It turns out that Justitia tracks the tail latency closely as long as *RefCount* is not huge. In Justitia, we set the default value of *RefCount* =10000 to have some memory of latency spikes but not longer enough to impact stable performance.

E Discussion

E.1 Why not simply use hardware priority queues in the RNIC?

Mellanox NICs have priority queues, but as we mention in the paper, the number of queues they support is very limited (e.g., only 2 lossless queues in the RoCE NICs we test out), and we have illustrated such limited number priority queues are insufficient to provide isolation in Figure 23. In addition, the time needed to reconfigure and modify the mapping from applications' QPs to the priority queues is in the order of milliseconds. Last but not the least, it is sometimes also desireable to provide isolation inside a priority level (e.g., bandwidth-sensitive applications and latency-sensitive applications are both assigned with the same QoS level) where hardware priority queues will not be sufficient. Thus, using the priority queues provided by existing hardware does not solve the isolation problem that Justitia faces.

E.2 Why use only 1 QP in most of the microbenchmark experiments?

We use a small number of QPs to show that the performance isolation issues can easily occur even with a very small number of active connections. We also test with more number of QPs but the results are placed in Appendix due to limit of space. In fact, adding more QPs exacerbates the performance degradation (Figure 30 in the appendix).

E.3 How does Justitia handle the incast experiment?

Justitia leverages receiver-side updates to make sure the correct minimum rate guarantees are updated correctly at each sender. Due to large latency spike in the case of a network incast, senders will mostly like send via the minimum guaranteed rate (R_{min}) given the latency target will not be met. We discussed receiver-side updates in Section 4.5 and illustrate Justitia complements with existing congestion control and can further help reduce receiver-side engine congestion in Section 6.5.

E.4 Does reference flow and receiver-side updates create additonal congestion in a large scale deployment?

The reference flow sends small messages (10 Bytes every 20 μs) and only amount to a very small Gbps number (1e6 / 20 * 10 / 1e9 * 8 = 0.004 Gbps), which consumes less than 0.1% of the total link capacity even at nodes with only 10 Gbps link, and thus is not likely to generate any hot spot in the network. When the server broadcasts the receiver-side update, the message is sent using SEND and RECV with a message size of 16 Bytes. With even 1000 client machines this amounts to around 16KB total message size, which is too small to create a potential congestion problem.

In the case of a large-scale latency-sensitive flow incast, if congestion indeed happens, DCQCN will work together with Justitia since it is the major congestion control dealing with fabric congestion. In this scenario, adding more latency-sensitive flows does not prevent Justitia guaranteeing bandwidth share of bandwidth hungry applications.

In the current design of Justitia, the bandwidth-sensitive applications can be rate-limited due to a coexisting latency-sensitive application which is launched at the same host but sends data to a different destination. This is intended behavior to mitigate the anomalies caused by contention at sender-side RNICs, which happens regardless of whether two competing applications are targeting the same receiver. We defer a comprehensive fabric-level solution which involves multiple senders and receives as our future work.

E.5 How to ensure all cooperating SW uses the right protocols and protocol versions?

To deploy Justitia, one only needs to install the Justitia Daemon code and a modified Mellanox driver code on the host machine, and Justitia is compatible with all existing RDMA protocols, including RDMA over Infiniband and RoCE. In datacenter deployments, cluster management tools like Ansible can be used to ensure the appropriate code is deployed at each machine. Additionally, it is straightforward to upgrade Justitia. Because each server in Justitia operates independently, it is not necessary for the same version of Justitia to be deployed across the cluster. Justitia will operate as long as servers are running some version of Justitia.

E.6 How can Justitia be implemented in hardware?

Without having a software layer to split the large RDMA operations before they arrive at the NIC, one probably need to somehow control how the NIC issues PCIe reads. Hardware is often optimized for performance, which in fact is why we are having such isolation issues, so simply decreasing the size of each PCIe reads will definitely affect its maximum throughput performance. To bring Justitia into the hardware design, similar to what we have done in the software layer, the hardware need to recognize when splitting and pacing is needed to provide isolation, and when it should process at

maximum capacity for higher utilization.

E.7 Long-term value of Justitia

As RNICs keep evolving, its performance isolation issues may be mitigated in newer hardware designs. The purpose of this work is to show that there exist such isolation issues in current kernel-bypass networks and illustrate one working approach to mitigate the issue. Design ideas presented in this work can inform hardware designers when developing future RNIC as well as programmable NIC designs.

F Future Research Directions

Interesting short-term improvements of this work include, among others, dynamically determining an application's performance requirements to handle multi-modal applications, handling idle applications, and extending to more complicated application- and/or tenant-level isolation isolation policies. Long-term future directions include implementing Justitia logic on an RNIC and integrating Justitia with congestion control algorithms.

We highlight these immediate next-steps in the following:

Dynamic Classification (Strategyproof Justitia). Applications may not always correctly or truthfully identify their flow types. To improve Justitia, it is possible to modify the driver to monitor QP usage and automatically identify whether individual connections are bandwidth-, throughput-, or latency-sensitive. This would provide support for multi-model applications.

Idle Applications. It is straightforward to support idle applications in Justitia without wasting bandwidth. If a bandwidth hungry app stops sending messages for a long time but does not exit, the driver and daemon can work together to stop token issuing when tokens are not being used and a configurable backlog of tokens has been accumulated.

Justitia at Application and Tenant Levels. Currently, Justitia isolates applications/tenants by treating all flows from the same originator as one logical flow with a single type. However, for an application with flows with different requirements or for a tenant running multiple applications competing with another tenant only running a single application, more complex policies may be desirable. With Justitia, it is straightforward to instead support per-tenant, per-application, or per-flow-group isolation. This is done by allocating tokens at multiple different granularities.

Co-Designing with Congestion Control. Although Justitia effectively complements DCQCN (§6.3) in simple scenarios, DCQCN considers only bandwidth-sensitive flows. A key future work would be a ground-up co-design of Justitia with DCQCN [64] or TIMELY [44] to handle all three traffic types for the entire fabric with sender- and receiver-side contentions (§6.5). While network calculus and service curves [12, 13, 29, 59] dealt with point-to-point bandwidth- and latency-sensitive flows, their straightforward usage can be limited by multi-resource RNICs and throughput-sensitive flows. At the fabric level, exploring a Fastpass-style centralized solution [48] can be another future work.