Harmony: A Congestion-free Datacenter Architecture

Saksham Agarwal
Cornell University

Qizhe Cai
Cornell University

Rachit Agarwal
Cornell University

David Shmoys
Cornell University

Amin Vahdat
Google

Abstract

Datacenter networks today provide best-effort delivery—messages may experience unpredictable queueing, delays, and drops due to switch buffer overflows within the network. Such weak guarantees reduce the set of assumptions that system designers can rely upon from the network, thus introducing inefficiency and complexity in host hardware and software.

We present Harmony, a datacenter network architecture that provides "congestion-free" message delivery guarantees—each message, once transmitted by the sender, experiences bounded queueing at each switch in the network. Thus, by design, Harmony ensures that network delays are bounded in failure-free scenarios, and that congestion-related drops are eliminated. We establish, both theoretically and empirically, that Harmony provides these powerful properties with near-zero overheads compared to best-effort delivery networks: it incurs a tiny additive latency overhead that diminishes with message sizes, and achieves near-optimal network utilization.

1 Introduction

Datacenter networks today provide best-effort delivery—it is hard, or even impossible, to bound queueing and delays experienced by messages at switches; even worse, messages may be dropped due to switch buffer overflows, and may need to be retransmitted multiple times before they are delivered to their destination. As a result, messages experience unpredictable and variable network delays and congestion-related drops. Such unpredictability and variability reduces the set of assumptions that system designers can rely upon from the network. Thus, to operate correctly on such best-effort delivery networks, host hardware and software must embrace inefficiency and complexity.

We present Harmony, a distributed packet-switched datacenter network architecture that provides powerful "congestion-free" message delivery guarantees: each message, once transmitted by the sender, is guaranteed to experience a small bounded amount of queueing at each switch along the path(s) it traverses. Thus, by design, Harmony ensures that network delays are bounded in failure-free scenarios, and that congestion-related drops are eliminated. The bounded network delay and zero switch buffer overflow guarantees also allow Harmony to handle inevitable failures efficiently: since delayed and/or undelivered messages are limited to hardware failures, Harmony enables extremely fast failure reaction.

Harmony enables these powerful properties by placing its intellectual roots in the classical Resource ReSerVation protocol (RSVP) [71]. We demonstrate that naïvely using RSVP on datacenter networks enables bounded queueing at each switch; however, throughput can be significantly lower than the optimal. Intuitively, the core reason for the suboptimal throughput is that the RSVP-based design maintains the invariant that each switch observes zero queueing at all times; enforcing zero queueing leads to non-trivial throughput overheads in distributed designs. Harmony combines the idea of virtual channels [23, 60] with RSVP to orchestrate network resources among competing messages while maintaining the invariant that each switch observes bounded, albeit potentially non-zero, queueing. We establish theoretically that allowing a small amount of queueing at the switches enables Harmony to provide bounded queueing and network delay guarantees while achieving near-optimal throughput.

We evaluate an end-to-end implementation of Harmony over a testbed, and over simulations across a variety of settings that mix-and-match multiple workloads, traffic patterns, network topologies, network oversubscription, and network loads with and without background (best-effort) traffic. Our evaluation reveals several interesting phenomenon. First, we find that Harmony achieves network delay and throughput very close to centralized zero-queue network designs [55], while providing the benefits of a completely distributed architecture. Second, rather surprisingly, we find that Harmony's tail latency and throughput is comparable or even better than state-of-the-art distributed datacenter protocols [19, 27, 32, 33, 44, 51] that provide best-effort delivery. Finally, we find that Harmony performance near-perfectly matches our theoretical bounds.

Harmony demonstrates that it is possible to rearchitect dat-

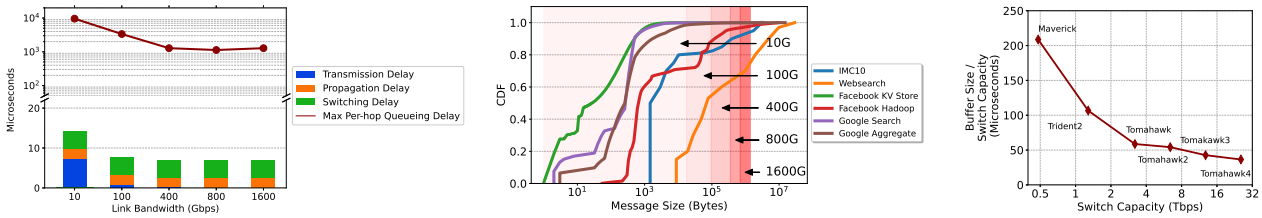


Figure 1: **Motivation for congestion-free datacenter architectures**. (left) modern high-bandwidth datacenter networks have tiny unloaded RTTs, and worst-case per-hop queueing delays can be orders-of-magnitude higher than unloaded RTTs; (center) a large fraction of messages are smaller than a single BDP for high-bandwidth networks, rendering modern best-effort delivery protocols inefficient; (right) switch buffer sizes are growing slower than switch capacities, resulting in increasingly more packet drops for best-effort delivery protocols. Discussion in §2.

acenter networks to provide much stronger guarantees than today's best-effort delivery networks, while maintaining near-optimal performance. There are two important caveats, however. First, our point is not that every application will benefit from Harmony—achieving these guarantees requires a tiny additive latency overhead (that diminishes with message sizes) at low and moderate loads; since not all applications may want to tradeoff such an overhead for strong guarantees from the network, datacenter networks should simultaneously support both Harmony and best-effort delivery, allowing applications to choose between the two depending on desirable goals. Second, the current implementation of Harmony requires support from the network: the two highest priority levels, a small amount of per-port soft state, support for embedding path identifiers in packet headers, routing based on path identifiers in packet headers, and support for packet modifications. While modern (programmable) switch hardware readily support these functionalities, it remains an interesting question whether Harmony guarantees can be achieved without network support. Finally, Harmony provides network-layer guarantees: bounded queueing, delays, and zero switch buffer overflows within the network; additional work is needed at each layer of systems stack to reap Harmony benefits in terms of improved host software and hardware. While it may take longer than the lifetime of a single project to realize systems that efficiently exploit all the benefits of Harmony, we believe the potential benefits make it a worthwhile exploration.

2 Congestion-free Datacenters: Motivation

Our exploration of congestion-free datacenter network architectures is motivated by three datacenter hardware trends[1].

Modern datacenter networks have tiny unloaded[2] round trip times (RTTs); queueing delays and buffer overflow are the root cause of unpredictability. Network hardware has improved over the past few years: multi-hundred gigabit links are already being deployed and Terabit Ethernet deployments are anticipated soon. Thus, datacenter networks will soon, if not already, support single-digit microsecond unloaded RTTs between hosts—as shown in Figure 1(left), transmission delays are reduced to a bare minimum, and unloaded RTTs are now sum of tiny propagation delays and switching delays. Furthermore, emergence and deployment of high-performance host network stacks [21, 38, 46], hardware-offloaded network stacks [2, 26, 58], host congestion control mechanisms [7, 31], and μs-scale host schedulers [21, 37, 53] have reduced host processing variability to a bare minimum. On the other hand, as shown in Figure 1(left), the worst-case queueing delay experienced by a packet is, and will continue to remain, much larger than unloaded RTTs. Put together, these trends result in queueing delays and buffer overflows as the root cause of network unpredictability.

Large bandwidth-delay products (BDPs) in modern datacenter networks make congestion control ineffective. Network RTTs limited by propagation and switching delays means that network BDP now increases linearly with link bandwidth. Such rapid increase in BDP means that most messages in the network now fit within a few BDPs. Figure 1(center) demonstrates this for several production workloads [8, 13, 18, 61, 65]: for 100 and 400Gbps links, more than 54% and 64% of the messages in the websearch workload are less than 1BDP; for the IMC10 workload, the corresponding numbers are 82% and 89%; a recent study from Google presents similar numbers for RPCs within Google [11]. Modern best-effort delivery datacenter transport protocols [19, 27, 32, 41, 44, 51, 72] advocate to blast the first BDP worth of packets into the network during the first RTT; blasting the first BDP worth of data fundamentally means that even optimal congestion response, one that detects congestion and responds perfectly in one network round trip, will not have time to converge to the "right" rates for an overwhelmingly large fraction of the messages resulting in larger amounts of data in flight, and larger unpredictability in queueing and network delays.

[1] We use Maverick (48-port 10G with 12MB buffers), Trident-2 (32-port 40G with 16MB buffers), Tomahawk (32-port 100G with 22MB buffers), Tomahawk-2 (64-port 100G with 42MB buffers), Tomahawk-3 (32-port 400G with 64MB buffers) and Tomahawk-4 (64-port 400G with 113MB buffers) switches [4].

[2] Defined as the maximum, across all sender-receiver pairs, time taken for a single MTU-sized packet to go from the sender to the receiver, and a 40-byte control packet to go from the receiver to the sender in the absence of any other packet in the network. This is a property of the network hardware.

Larger BDPs and relatively stagnant switch buffer sizes make packet drops more likely for best-effort delivery networks. Rapid increase in BDP and switch capacities coupled with relatively stagnant switch buffer sizes (Figure 1(right)) also suggest that increasing link bandwidths will make it easier to overwhelm switch buffers. As a result, best-effort delivery networks will experience an increasingly larger fraction of packets dropped within the network, resulting in even higher network unpredictability and reduced network throughput.

3 Harmony

This section presents Harmony, a distributed packet-switched datacenter architecture that guarantees bounded queueing at each switch in the network, while achieving near-optimal network throughput. We describe the Harmony protocol in §3.1, provide low-level details on Harmony design in §3.2, and establish theoretical properties of Harmony in §3.3.

3.1 Harmony Protocol

Harmony ensures bounded queueing at each switch by placing its intellectual roots in the classical Resource ReSerVation protocol (RSVP) [70, 71], and integrating it with another classical idea: virtual channels [23, 60].

We outline the Harmony protocol below, followed by an intuitive description of how it guarantees bounded queueing while achieving near-optimal throughput. Harmony protocol uses the following two constructs: host slots and virtual links.

Host slots. Each sender and receiver maintains K slots, each allowing a message to be sent and received using bandwidth B/K, where B is the access link bandwidth. Importantly, each slot can be allocated to at most one message; however, a message may be allocated more than one slot (that is, a sender-receiver pair may simultaneously use as many as K slots to send and receive a message).

Virtual links. Each physical link in the network is logically decomposed into virtual links, each of bandwidth B/K. Each virtual link can be allocated to at most one message, allowing the message to be transmitted using bandwidth B/K; however, a message may be allocated more than one virtual link.

Harmony protocol. The Harmony protocol is extremely simple, and works as follows:

- Each sender, immediately upon a message arrival, sends a `request` control packet to the receiver.
- Each receiver keeps track of its free slots. Upon receiving a `request`, if the receiver has one or more free slots, it assigns one of the slots to the message, updates the number of free slots, and sends a `rsvp` to the corresponding sender. If the receiver has no free slots, it adds the `request` to a list of `pending_requests` and starts a `timer` (using its local clock) for the `request`. Whenever a slot becomes free, the receiver picks the `request` with the largest `timer` value, assigns the slot to the message, updates its number of free slots, and sends a `rsvp` to the sender for this `request`.
- Whenever the `timer` for a `request` reaches $\delta_{\text{admission}}$ time, the receiver removes the `request` from the `pending_requests` list and sends a `reject` to the sender indicating that the receiver is unable to admit the message due to high load at the receiver.
- Each switch maintains, for each of its links, the number of free virtual links. Upon receiving a `rsvp`, the switch uniform randomly chooses a free outgoing virtual link along one of the shortest paths to the sender, embeds its identifier into the `rsvp` header, updates the number of free virtual links for the port, and forwards the `rsvp` on to that link. If no free virtual link is available, the switch transforms the `rsvp` into a `reject` packet, and sends it towards both the sender and the receiver using information in `rsvp` headers. `reject` traverses the same set of switches that forwarded `rsvp` so that corresponding links can be freed. Importantly, only shortest paths are used to ensure deadlock freedom.
- Each sender also keeps track of its free slots. Upon receiving a `rsvp`, if the sender has one or more free slots, it assigns one of the slots to the message, updates the number of free slots, and starts transmitting the message at a slot bandwidth. Once the message is finished, the sender sends a `complete` and marks the corresponding slot as free. Upon receiving a `rsvp`, if no free slots are available, the sender sends a `reject` to the receiver. All data and control packets are source routed using switch identifiers in the `rsvp` header.
- Each switch, upon receiving the `reject` or `complete`, marks the corresponding virtual link as free (using identifiers in the header), and forwards it to the next hop;
- Each receiver, upon receiving `reject` or `complete`, marks one of its slots as free; for `complete`, the receiver also sends a `complete` to the sender indicating completion of message transmission.
- Each sender, upon receiving a `reject` or `complete`, marks such for the message.

We describe additional design details for Harmony, including fast failure reaction, handling background (best-effort) delivery traffic, prioritization mechanisms for isolating messages from control packets and best-effort delivery traffic, optimizations like multi-slot allocation to individual messages, etc., in the next subsection. Below, we provide an intuitive description on how Harmony guarantees bounded queue at each switch and how it uses multiple virtual links and host slots to achieve near-optimal throughput. Figure 2 shows an example.

Intuitively understanding the invariants maintained by Harmony. Harmony maintains two invariants. First, for each outgoing link at each switch, the sum of arrival rates of all messages to be transmitted on the outgoing link is no more than the link bandwidth; and second, at all times, a small bounded

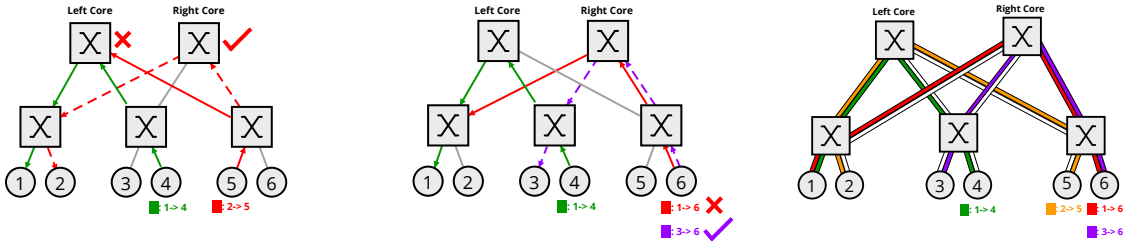


Figure 2: **Understanding benefits of virtual links and host slots. (left)** without multiple virtual links, network may be underutilized due to switches making uncoordinated decisions; and **(center)** without multiple host slots, network may be underutilized due to hosts making uncoordinated decisions; **(right)** using multiple virtual links and host slots, Harmony alleviates network underutilization. Discussion in §3.1.

number of messages use any link in the network. The first invariant ensures zero persistent queueing since the rate at which messages (destined to a particular outgoing link) arrive is at most the link bandwidth. However, while each message uses a different virtual link, multiple messages may now share a physical link; thus, data from multiple messages may arrive at the switch (via different virtual links) at the same time resulting in transient queueing. The second invariant ensures that transient queueing is bounded since a physical link is shared by at most as many messages as the number of virtual links (in §3.3, we will bound worst-case queueing in terms of link bandwidth, number of virtual links and maximum number of switches along any network path). Put together, these two invariants are sufficient for Harmony to guarantee bounded queueing at each switch in the network.

Intuitively understanding the necessity of virtual links. Virtual links in Harmony design are not merely a heuristic—they are key to Harmony achieving near-optimal throughput. Intuitively, in the above design, switches make decisions on forwarding `rsvp` based on purely local information, without any view of the state of links at neighboring switches. Switches making such uncoordinated decisions could lead to requests being rejected even if there is a path available in the network. For instance, consider the example shown in Figure 2(left): here, green message is using a reserved path $1 \rightarrow 4$; when receiver 5 sends a `rsvp` toward 2, its leaf switch uniform randomly chooses the left spine switch and forwards the `rsvp`. Since there is no unreserved outgoing link from the left spine switch to 2, the `rsvp` will be rejected. Had the leaf chosen the right spine switch, it would have successfully reached sender 2, improving network throughput. As shown in Figure 2(right), virtual links help alleviate unnecessary `rsvp` rejections by enabling fine-grained sharing of network resources. Intuitively, since an `rsvp` message can be forwarded along any of the virtual links, and there are more virtual links than physical links, the probability of an `rsvp` being unnecessarily rejected reduces significantly.

Intuitively understanding the necessity of host slots. Host slots, on the other hand, help alleviate network underutilization due to hosts making uncoordinated decisions. Specifically, since receivers do not have information about the network and/or sender state, an incorrect choice on messages to send `rsvp` for can lead to suboptimality: for the last `rsvp` sent by the receiver, it may receive a `reject` if there is no available path to the sender. This, in turn, will result in suboptimal throughput. An example is shown in Figure 2(center): here, receiver 6 has two outstanding requests, one from sender 1 (red) and one from sender 3 (violet). The receiver, however, can choose only one message to send `rsvp` to; if the receiver sends a `request` for 1, it will receive a `reject` since there is no unreserved path between 1 and 6. However, ideally the receiver should have selected 3 for `request` since there is an unreserved path between 3 and 6. As shown in Figure 2(right), with multiple host slots, receivers can now send `rsvp` for multiple messages again increasing the network throughput.

Intuitively understanding the benefits of $\delta_{admission}$. Harmony design allows requests to wait for time $\delta_{\text{admission}}$ at the receiver before they can be rejected. This parameter enables Harmony to handle bursty traffic and/or incasts—with higher $\delta_{\text{admission}}$ values, Harmony will be able to admit larger amounts of bursty traffic and/or larger incasts. Network operator can choose the right $\delta_{\text{admission}}$ to accommodate the desired bursty traffic and/or incast within Harmony.

3.2 Harmony Design Details

We now provide additional details on Harmony design.

Handling control packets. Harmony design uses a number of control packets—{`request`, `rsvp`, `reject`, `complete`}. These control packets utilize non-zero bandwidth, and could potentially interfere with message transmission at switches. To minimize impact of control packets on message transmissions, Harmony uses priorities, a standard technique for traffic isolation in modern datacenters [19, 27, 64]. Specifically, all messages are transmitted using the highest priority, all `rsvp` and `complete` messages are also transmitted using the highest priority, and all other control packets are transmitted using the second highest priority. In [5], we provide details on how this prioritization mechanism maintains Harmony queueing and delay bounds; intuitively, `rsvp` and `complete` packets transmitted using the highest priority does not impact message queueing much: these are tiny packets, and each link forwards a small number of `rsvp` and `complete` packets

(in the worst-case, as many as the number of virtual links). Furthermore, since Harmony bounds the switch buffer occupancy due to data packets at all times, and since control packets are small, Harmony can absorb large number of low-priority control packets within switch buffers—for instance, $16-32$MB buffer space in switches today can hold up to $\sim 400,000-800,000$ concurrent 40-byte control packets per switch; even with Harmony's worst-case queueing bound for data packets, we can still sustain $\sim 300,000-700,000$ concurrent control packets per switch. To ensure that low-priority control packets are not starved by data packets, Harmony also reserves a small amount of bandwidth at each host and switch for control packet transmission.

Exploiting bounded network delays in failure-free scenarios for fast reaction to failures and `reject`*s.* Harmony eliminates congestion-related drops in failure-free scenarios. However, inevitable hardware failures can still lead to data and control packet drops. Harmony uses the key insight that all data packets in Harmony traversing a bounded-queue path means that receivers know exactly when to expect data packets corresponding to a `rsvp` message in a failure-free scenario; similarly, senders know exactly when to expect `complete` or `rsvp` message in a failure-free scenario. Indeed, absence of any of these events must imply a hardware failure or a `reject`. Harmony exploits such a predictability in failure-free scenarios to trigger fast retransmission mechanism—as soon as a sender or a receiver infers a failure or a `reject` has happened, they trigger fast retransmission (receiver sending a `failure` control packet to the sender, and/or sender retransmitting data using a new `request` or using best-effort delivery interface, as described below).

Exploiting bounded network delays in failure-free scenarios for efficient multi-slot allocation to individual messages. To ensure high utilization in practice, Harmony assigns multiple host slots and virtual links to a single message when a receiver has less than K active messages. A message that has been allocated at least one slot is marked as an active message. Whenever receiver has a free slot and no pending requests, an active message is allocated additional slots; to ensure that additional slot allocation does not block future requests, additional slots are allocated for one BDP worth of data only. Harmony receivers always send one `rsvp` per slot, independent of which message the `rsvp` is for. When a new `rsvp` message is received by the sender for an active message, an additional virtual path of bandwidth B/K has been reserved and the sender can now send the message using an additional slot (increasing the transmission rate for this message by B/K). This ensures correctness, while also maximizing network throughput in practice [5].

Exploiting bounded network delays in failure-free scenarios for efficiently handling best-effort delivery traffic. Harmony can be integrated with any existing datacenter transport mechanism to handle best-effort traffic. The only constraint is that bounded-queue traffic and control packets must be isolated from best-effort traffic; Harmony thus uses the third priority level for best-effort traffic. Thus, the only impact best-effort traffic has on Harmony's timely delivery or bounded queueing properties is that latency can increase by as many packet transmission times as the number of switches in the path—each switch can incur one extra packet transmission time due to non-preemptive nature of today's switches (Harmony packet will be transmitted immediately after ongoing transmission of best-effort traffic packet).

We present additional details on Harmony design such as handling drops of control packets under failures, utilizing predictable network delays to optimize Harmony performance via pipelining `rsvp` packets, handling packet reordering, etc., in the technical report [5].

3.3 Harmony Theoretical Properties

We now establish theoretical properties of Harmony under a model similar to previous studies [19, 22, 45, 47, 55, 62].

Network model. For our theoretical analysis, we assume full-bisection bandwidth datacenter networks; this assumption is purely for ease of theoretical analysis—Harmony guarantees bounded queueing even for oversubscribed networks [5]. We model the network as a $N \times N$ crossbar switch, with N inputs (sender hosts) and N outputs (receiver hosts). Each input has a buffer of infinite capacity, and is partitioned into N virtual output queues (VOQ); each output also has a buffer of infinite size. The virtual output queue VOQ_{ij} holds packets arriving at input i and are destined for output j. It is well-known that, under the above assumptions, datacenter network transfers can be modeled as a bipartite matching problem [9, 19, 27, 55].

We assume that time is slotted. Packets are transferred from input buffers to output buffers in scheduling cycles. Each scheduling cycle consists of a matching phase and a data transfer phase. In the matching phase, an algorithm computes a matching between inputs and outputs in a manner that no input may be matched to more than one output, and no output is matched to more than one input. For each input i and output j matched to each other, one MTU-sized packet is removed from VOQ_{ij} and one MTU-sized packet is put into the output buffer of j during the data transfer phase (assuming VOQ_{ij} is non-empty). The network is said to have a speedup of s if, during each time slot, there are s scheduling cycles.

Let the average rate of packet arrival at VOQ_{ij} be λ_{ij}. The input traffic is said to be admissible if $\sum_i \lambda_{ij} < 1$ and $\sum_j \lambda_{ij} < 1$, that is, the load at each input and output is less than 1.

Fundamental limits. It is known that full-bisection bandwidth networks can achieve 100% throughput for any admissible input traffic as long as the underlying protocol computes a so-called maximum matching [22]. However, computing a maximum matching typically requires complex implementations; thus, existing switch fabrics and network protocols typically compute a so-called maximal matching [19, 27, 55].

A classical result establishes that any maximal matching based protocol requires a network with a speed up of 2 to achieve 100% throughput [22, 45] for all admissible input traffic.

Theoretical analysis for Harmony. We now provide bounds on queueing, network delays and throughput for Harmony.

Theorem 3.1 *Let #H be the number of switches along the longest path (across all sender-receiver pairs), K be the number of virtual links per physical link, B be the minimum bandwidth across physical links, and p be the maximum packet size. Then, maximum queueing observed by any packet at any switch in Harmony is bounded by*

$$Q_{Harmony} \leq \#H \cdot (K-1) \cdot p$$

Moreover, the total queueing delay (across all switches) incurred by any packet in Harmony is bounded by:

$$\delta_{queueing} \leq \frac{\#H \cdot (\#H+1) \cdot (K-1)}{2} \cdot \frac{p}{B} \qquad (1)$$

We provide the full proof in [5]. Intuitively, given that we have K virtual links sharing the physical link, we would expect a queue bound of $(K-1) \cdot p$; however, our proof demonstrates that, in the worst-case, packet transmissions across multiple switches can (mis)align in a manner that the worst-case queueing bound worsens with more switches along the path. The bound on queueing delay follows by aggregating the precise amount of queueing delay at each switch along the path. These bounds hold independent of the underlying network topology (since they are parameterized by number of switches along the path and K).

Let $S^\star$ be the minimum buffer size across all switches, let $\#\ell$ be the number of ports at the switch, and let $K^\star = S^\star/(\#H \cdot p \cdot \#\ell)$. Then, the first part of the theorem shows that for $K \leq K^\star$, Harmony will ensure that queueing at each switch never grows beyond the switch buffer size, and that congestion-related drops are eliminated. For modern datacenter hardware, $K^\star$ turns out to be large enough for all practical purposes. For instance, for a datacenter organized around a FatTree topology using 100Gbps Tomahawk switches (32 ports and 22MB switch buffer size) and with 1.5KB packet sizes, we get $K^\star \approx 91$. For a FatTree topology with 400Gbps Tomahawk-4 switches (64 ports and 113MB buffers), we get $K^\star = 235$.

Theorem 3.1 not only guarantees that congestion-related drops are eliminated for $K \leq K^\star$, but also bounds the queueing delay seen by any packet: for the above two topologies and for $K = 8$, we get that $\delta_{\text{queueing}} = 12.6\mu$s and $\delta_{\text{queueing}} = 3.15\mu$s, respectively. Thus, Harmony guarantees tiny worst-case queueing delays. The bound on queueing delays can be trivially used to bound the delay between the sender starting to transmit a message m (upon receiving an `rsvp`) until it receives the `complete` for the message [5]. Extending Harmony design to guarantee end-to-end delays (that is, including processing delays at the sender and at the receiver, and queueing delays at the sender) is an intriguing open question (§6).

Harmony uses a distributed protocol for allocating host slots and virtual links to senders and receivers, with hosts and switches making decisions based on purely local information. It, thus, becomes an interesting question to characterize the quality of successful allocations—requests for which the receiver sends a `rsvp` that is not rejected within the network. The following theorem characterizes the slot and virtual link allocation efficiency of Harmony for a full-bisection bandwidth leaf-spine topology with respect to an ideal centralized algorithm that performs K maximal matching for allocating host slots, and perfectly allocates virtual links to matched slots (see [5] for a basic extension of [55] that realizes such an ideal centralized algorithm). Let $\Theta^\star$ denote the number of successful slot allocations by such an ideal algorithm.

Theorem 3.2 *Let K be the number of virtual links per physical link and C be the number of core switches in the topology. Then, the expected number of successful slot allocations by Harmony, randomized over choice of* `rsvp` *forwarding decisions, is:*

$$\mathbb{E}[\Theta_{Harmony}] = f(K,C) \cdot \Theta^\star, \qquad \textit{where,} \qquad (2)$$

$$f(K,C) = \frac{\sum_{i=0}^{KC} \min(i,K) \binom{KC}{i} \left(\frac{1}{C}\right)^i \left(1-\frac{1}{C}\right)^{(KC-i)}}{K}$$

We provide a full proof in Harmony technical report [5]. At a high-level, our proof for Theorem 3.2 uses an argument similar to the classical balls-and-bins problem [59] to establish the slot allocation efficiency in Harmony. This result, similar to most prior analytical results [19, 55], assumes two-tier full-bisection bandwidth leaf-spine network topology; it is an intriguing open question to generalize our bounds to FatTree, expander-based and oversubscribed network topologies. Nevertheless, this result provides us several insights on the behavior of Harmony. First, for the special case of $K = 1$ (which is roughly equivalent to RSVP adapted to datacenter networks), the above expression leads to $\mathbb{E}[\Theta_{\text{Harmony}|K=1}] = (1-1/e)\Theta^\star$, where $e \approx 2.72$ is base of the natural logarithm. That is, for $K = 1$, the slot allocation efficiency is merely $\sim$63% of the optimal. Second, Figure 4 confirms the intuition discussed earlier: increasing the number of virtual links and host slots (increasing K) results in significantly improved efficiency of Harmony, converging to the optimal $\Theta^\star$. Harmony evaluation in §5.1 demonstrates that, in practice, Harmony achieves significantly better performance even for small values of K.

4 Harmony Implementation

We have done prototype implementation of Harmony using readily available programmable switches and DPDK-based hosts. This section provides some of the most interesting implementation details. We provide more details in [5].

The Harmony interface. Harmony currently implements a RPC interface [38, 66]: messages are submitted to a submission queue and the message delivery status is updated in

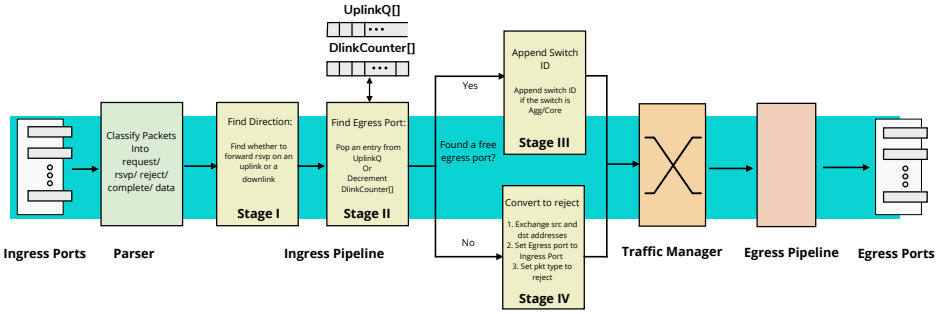


Figure 3: Harmony can be realized using programmable switches supporting PSA architecture. More description on §4.

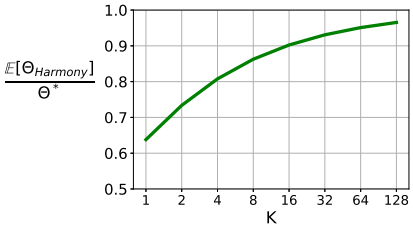


Figure 4: Harmony's performance in terms of successful host slot allocations—$\mathbb{E}[\Theta_{Harmony}]$—converges to that of an ideal centralized maximal matching based algorithm with increasing values of K. We use a 64-port leaf-spine full-bisection bandwidth topology with 32 core switches in this figure.

the completion queue (using a `complete` or `failure` flag). The only difference between Harmony and standard RPC interfaces is that the completion queue in Harmony has one additional flag: `reject`, which indicates that Harmony is unable to guarantee bounded queueing for that message (*e.g.*, due to failures, high receiver load, high network load, etc.). If the flag is either `reject` or `failure`, the message can be resubmitted using the Harmony interface, or using the standard best-effort delivery interface.

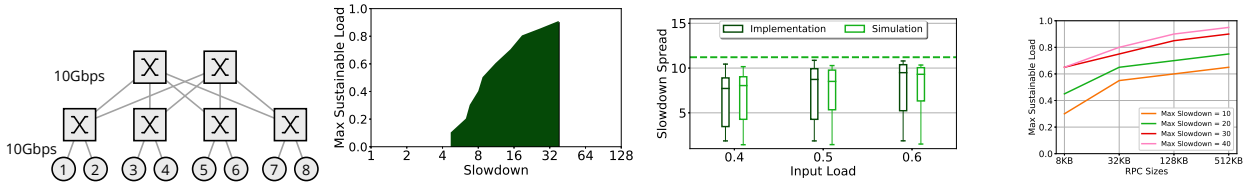
Harmony host implementation. Harmony maintains the invariant of bounded queueing at switches thus alleviating the need for congestion control on the fast path at hosts; as a result, Harmony can be easily integrated with existing high-performance userspace or in-kernel network stacks [21,38,46], as well as existing accelerator based network stacks [2,26,58]. To implement Harmony-specific functionality, much of our host implementation uses modules from existing network stacks. For example, generating and responding to `rsvp` messages is similar to the grant mechanism in receiver-driven transport protocols [19, 27, 32, 33, 51]. As another example, Harmony design requires senders to transmit messages at a rate that is dependent on the number of slots allocated to that message; such a rate limit functionality at the sender is already implemented in almost all network stacks. Overall, our Harmony host implementation uses ∼3107 lines of code.

Harmony switch implementation. Harmony switch implementation uses programmable switches [3]. The data plane of these switches employs Portable Switch Architecture [3], and is composed of a parser, an ingress and an egress pipeline and a traffic manager (Figure 3). Upon a packet arrival on an ingress port, Harmony parser extracts the header, identifies the packet type (`rsvp`, `reject`, `complete`, or data packet), and forwards the packet to the ingress pipeline. The ingress pipeline decides the egress port to forward the packet to, embeds switch identifiers within the packet header (using techniques from path tracing mechanisms on programmable switches [67, 68]), and then passes the packet to the traffic manager (which then forwards the packet to the desired egress port). Harmony's implementation of the ingress pipeline uses two data structures: A FIFO `UplinkQ`, and a counter `DlinkCounter`. `UplinkQ` maintains the available virtual links which can be reserved by `rsvp` packets while traversing uplink (toward the spine, or the core switch). `DlinkCounter` maintains the number of reserved virtual links at the downlink path (away from the core switches). In addition, Harmony switches maintain a small constant amount of state to avoid blocking of virtual links during control packet drops [5].

5 Harmony Evaluation

We now evaluate Harmony performance over a small-scale testbed (§5.1), and over large-scale packet-level simulations (§5.2). We start by describing our evaluation setup.

Network Topologies. We use a testbed with 8 servers organized along a two-tier topology with 10Gbps links (as shown in Figure 5a). For our simulations, we use the same default setup as in [9, 19, 27, 44]: the standard 144-node leaf-spine topology with 9 top-of-rack (ToR) switches, each connected to 16 hosts with 100Gbps access link bandwidth. The propagation and switching delay are 200ns and 450ns, respectively. We use switch buffer capacity of 32MB. The unloaded RTTs and bandwidth-delay product values for this topology are 4.6μs and 56KB, respectively. We also use oversubscribed and FatTree topologies. For the former, we vary the oversubscription (1:1 to 2:1 and to 4:1) in our leaf-spine topology by correspondingly reducing the leaf-spine link bandwidths; we reduce the load based on oversubscription factor to ensure that the traffic remains admissible. Our three-tier FatTree topol-



(a) Testbed Topology (b) Feasible region (c) Slowdown Spreads (d) Varying RPC Sizes

Figure 5: **Evaluation of Harmony implementation on a small-scale testbed.** (a) The testbed used in this evaluation; (b) Harmony's feasibility region in terms of achievable delays and maximum sustainable loads (the green dotted line represents Harmony's delay bound for this topology); (c) Harmony evaluation over the testbed almost perfectly matches the theoretical bounds, as well as simulation results in Harmony's packet-level simulator; (d) for a fixed delay bound, Harmony sustains larger loads with larger RPC sizes (as expected). Discussion in §5.1.

ogy contains 1024 hosts, 64 core switches, 128 aggregation switches, 128 ToRs and 100Gbps access link bandwidth. We use the same propagation delay, switching delay and switch buffer size as above. The unloaded RTT and bandwidth-delay product for this topology are 7.6μs and 93KB, respectively.

Evaluated schemes. Existing distributed datacenter transport protocols are not designed to achieve congestion-free communication; thus, comparison with Harmony would be unfair to any choice of protocols. Nevertheless, we evaluate Harmony performance against state-of-the-art sender-driven (HPCC [44]) and receiver-driven (dcPIM [19]) datacenter transport protocols. We use the default parameters from HPCC and dcPIM papers. We also extend the centralized Fastpass scheduling algorithm to support message-level scheduling[3] to provide an ideal baseline for Harmony. Unless specified otherwise, we use the following values for Harmony parameters: $K = 8$ and $\delta_{\text{admission}} = \sim 63\mu$s (a factor 5 of Harmony worst-case queueing delay bound); we also perform sensitivity analysis against these two Harmony parameters.

Workloads and traffic patterns. We used 128KB RPCs (as in [49, 72]) as our default workload since it allows us to provide in-depth insights due to each message having the same latency bound, but also present results for RPC sizes varying from 0.5KB to 512KB. For evaluation of Harmony with background (best-effort) traffic, we use a mix of RPCs (for traffic desiring congestion-free guarantees), and the standard web-search datacenter workload [9, 19, 27, 44] (for the best-effort traffic). We use the standard methodology of generating message arrival times using Poisson arrival process [8,9,19,27,44] and use an all-to-all traffic pattern by default. To avoid degenerate scenarios, we ensure that the generated load for each sender and each receiver is less than 1 for every $\delta_{\text{admission}}$ duration of time.

Evaluation metrics. Harmony guarantees bounded queueing at each switch in the network, thus ensuring bounded queueing delays for each message. However, Harmony does not bound processing delays at the sender and at the receiver, and queueing delays at the sender. Nevertheless, in our evaluation, we use Harmony queueing delay bound as the overall end-to-end bound. We use two metrics—maximum sustainable load (defined as the input load for which no message is rejected) and latency. For latency, we use the standard slowdown metric, defined as the ratio of the message completion time under loaded and unloaded scenarios; we present slowdown spread—{minimum, mean, 99%-ile, 99.9%-ile, and maximum}—using lower whisker, lower box edge, mid-line in the box, upper box edge, and upper whisker, respectively.

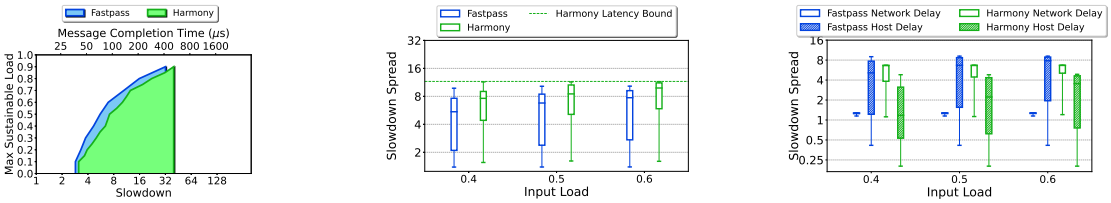
[3]The original Fastpass algorithm performs per-packet scheduling; this provides delay guarantees for each individual packet, but not for each message as in Harmony. We make two extensions: (a) to enable message-level delay guarantees, we ensure that all packets for a message are scheduled in consecutive timeslots; and (b) to enable a fair comparison in terms of maximum sustainable load, we allow a message request to wait for $\delta_{\text{admission}}$ time before it is rejected.

5.1 Harmony Testbed Evaluation

We start with evaluation of Harmony implementation over a small-scale testbed. Our testbed results are primarily to demonstrate the feasibility of Harmony—it is an extremely simplified setup with no contending applications, dedicated cores to ensure minimal host processing delays, no host congestion, etc.; more work is needed to evaluate end-to-end performance of Harmony over large-scale deployments.

Our first result, shown as the shaded region in Figure 5b, shows that Harmony enables a unique trade-off space between the delay bound and maximum sustainable load. Specifically, the two parameters in Harmony—K and $\delta_{\text{admission}}$—allow it to tradeoff the delay bound for higher maximum sustainable loads. Here, we vary K and $\delta_{\text{admission}}$, and plot the pareto curve on the observed worst-case delay and sustainable loads. This tradeoff essentially follows the intuition discussed earlier—with a larger bound on target message delay (via increasing K and $\delta_{\text{admission}}$), Harmony sustains higher loads. Figure 5d shows this tradeoff for varying RPC sizes: we observe that Harmony can sustain as high as 0.9 load for 128KB RPCs, and 0.65 load for 8KB RPCs.

We also use our testbed to verify the simulator fidelity; in particular, we incorporate the measured testbed parameters (link propagation, switching and average PCIe delays) into our simulator and show the corresponding results alongside the testbed results in Figure 5c. The testbed latency results near-perfectly match our simulator (which, in turn, matches the analytical bounds in Theorem 3.1 and Theorem 3.2).



(a) Feasible region (b) Slowdown spreads (c) Harmony latency breakdown

Figure 6: **Evaluation of Harmony and Fastpass using large-scale simulations**. Harmony achieves feasibility region surprisingly similar to Fastpass—for any given latency bound, it can sustain loads as high as Fastpass (and vice versa). Discussion in §5.2.

5.2 Harmony Large-Scale Simulation

We now evaluate Harmony over simulations, both in absence (Figure 6) and in presence (Figure 7) of best-effort delivery traffic. For the former, we compare Harmony with Fastpass; for the latter, we compare Harmony with HPCC and dcPIM.

[Figure 6, no best-effort delivery traffic]. Figure 6a shows maximum sustainable loads for varying target delay bound (or, slowdown) for both Harmony and Fastpass; each point in the feasibility region of Figure 6a is achieved using a combination of K and $\delta_{\text{admission}}$.

We observe that, despite using a purely distributed design, Harmony's feasibility region closely matches that of Fastpass. This result may be surprising. Digging deeper, we found that this result is due to Fastpass and Harmony making significantly different tradeoffs. Fastpass enforces zero queueing at each switch by transmitting each message on a dedicated pre-reserved path—the arrival rate for each message thus perfectly matches the outgoing bandwidth available for that message at each switch. However, this invariant—enforcing zero queueing at each network switch—incurs high overheads for a distributed protocol like Harmony. Instead, the key insight in Harmony design is that it is feasible to achieve bounded (but not necessarily zero) queueing at each switch using a completely distributed design, while maintaining high throughput. Figure 6c provides more insights on how Harmony achieves performance so close to an ideal centralized architecture even with non-zero queueing at switches. Essentially, by enabling bounded queueing (rather than zero queueing) within the network, Harmony significantly reduces the amount of time a message spends waiting at the sender before it can be scheduled and can start transmitting packets; this reduced host-side delay easily compensates for the increased queueing delay in the network when compared to Fastpass. Overall, this turns out to be a good tradeoff—Harmony, despite its distributed nature, achieves delays and maximum sustainable loads comparable to an ideal centralized datacenter architecture.

Harmony does make a different tradeoff compared to Fastpass: as shown in Figure 6b, it achieves $\sim 2\times$ higher mean and $\sim 1.3\times$ higher P99 latencies, while enjoying the benefits of a distributed datacenter architecture.

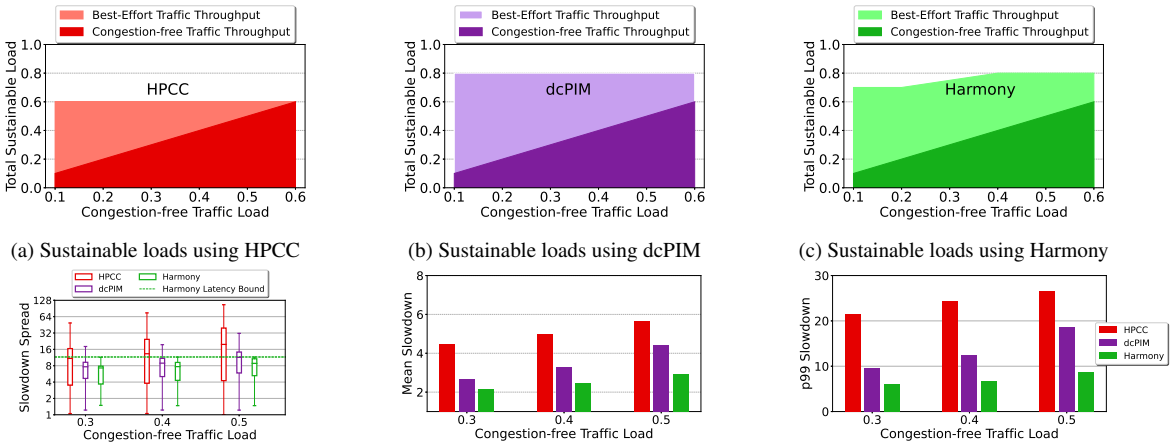
[Figure 7, Harmony performance with best-effort traffic]. We now discuss Harmony performance for scenarios where congestion-free traffic (using 128KB RPCs) coexists with best-effort delivery traffic (generated using websearch workload). We vary the load of congestion-free traffic from 0.3 to 0.5, and respectively vary the load of co-existing best-effort delivery traffic from 0.3 to 0.1, such that the total offered load is 0.6 and hence remains sustainable for all protocols.

Figure 7 shows a surprising result: when compared to existing state-of-the-art sender-based and receiver-based best-effort delivery protocols, Harmony sustains similar or higher loads for best-effort traffic (with varying congestion-free traffic load, as shown in top row), achieves significantly better tail latencies for congestion-free traffic, and significantly better mean and tail latencies for best-effort traffic! Upon digging deeper into this surprising result, we found the following two interesting phenomena.

First, as discussed earlier, Harmony ensures that the rate at which congestion-free traffic arrives at any switch perfectly matches the bandwidth available to forward this traffic; as a result, congestion-free traffic incurs a small amount of queueing at each switch. The result is that, despite lower priority, best-effort delivery traffic experiences minimal contention with congestion-free traffic in Harmony. For other protocols, however, these two traffic create queueing for each other resulting in lower sustainable loads and higher latencies. The problem exacerbates at higher loads.

Second, for HPCC, we observed that RPC traffic creates congestion in the network core despite a full-bisection bandwidth topology. This is because the transmission time for a 128KB RPC on a 100Gbps link is $\sim 10.2\mu$s, which is $\sim 2.2\times$ the unloaded network RTT. Thus, even a small amount of queueing can result in congestion control being ineffective—by the time congestion signal arrives, the network condition has changed. For dcPIM, we found that 128KB RPCs were larger than a single BDP and thus required matching before they can start transmitting; this leads to larger mean and tail latencies. Harmony avoids both of these problems—it ensures congestion-free network for the RPC traffic, and minimizes queueing experienced by best-effort delivery traffic.

[Figure 8, Sensitivity analysis for Harmony]. Figure 8 demonstrates that Harmony performance is robust across all evaluated scenarios—as long as the load is sustainable, Harmony achieves low tail slowdown and bounded worst-case message delays. We discuss three important observations.



(a) Sustainable loads using HPCC

(b) Sustainable loads using dcPIM

(c) Sustainable loads using Harmony

(d) Slowdown spread (Congestion-free traffic)

(e) Mean slowdowns (Best-effort delivery traffic)

(f) 99p slowdowns (Best-effort delivery traffic)

Figure 7: **Evaluation of Harmony with background (best-effort delivery) traffic.** Figures in the top row show maximum sustainable loads for traffic desiring best-effort delivery (lighter color) and traffic desiring congestion-free guarantees (darker color) for HPCC, dcPIM and Harmony. Bottom row shows the slowdown spread for congestion-free traffic (left), mean latency for best-effort traffic (center) and tail latency for best-effort traffic (right).

First, as expected, Harmony sustains low load for small-sized RPCs; for instance, Figure 8a shows that when RPC size is 4KB (that is, each RPC is $\sim$14$\times$ smaller than the network BDP), Harmony requires a target delay that is 20$\times$ larger than the unloaded RTT to sustain 0.5 load. For such small RPCs, modern best-effort delivery datacenter transport protocols will simply send all the packets within the first RTT, rendering congestion control essentially irrelevant. As a result, depending on the network load, network topology, oversubscription ratios, load balancing mechanisms and/or traffic patterns, queueing delays may or may not dominate network delays; thus, Harmony may or may not be the best choice for such small transfers. Recent studies from datacenter networks suggest that RPCs are much larger in practice (*e.g.*, a recent study from production datacenters demonstrates that median RPC size is greater than 40KB [11]; similar observations have been made in prior studies [49, 72]); for such real-world scenarios, Harmony guarantees small worst-case delays while sustaining high loads.

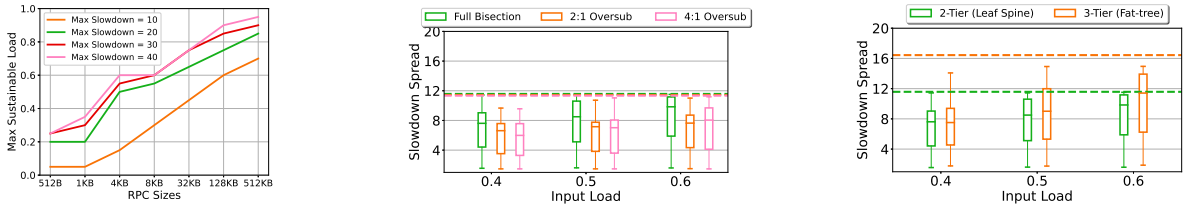
Second, Harmony maintains its performance for oversubscribed topologies. This is not surprising—as discussed in §3.3, for all sustainable loads, Harmony delay bounds hold even for oversubscribed topologies. This is because each message is still transmitted over a dedicated set of virtual links. Moreover, the worst-case queueing delay bounds remain near-identical with increasing oversubscription because (i) the maximum possible per-hop queueing decreases; and, (ii) the per-packet transmission delay increases, by roughly the same amount (the delay bound now uses the minimum link bandwidth in the topology). These two factors essentially balance out resulting in near-identical delay bound. Finally, since load is now defined with respect to the minimum link bandwidth in the topology, Harmony is also able to sustain similar loads as in full-bisection bandwidth topologies.

We present results for sensitivity analysis of Harmony performance over the two Harmony parameters—K and $\delta_{\text{admission}}$—in [5]. To briefly summarize, the results confirm the intuition from our theoretical analysis: with increase in K, Harmony delay bounds are higher but maximum sustainable load is also higher. Increase in $\delta_{\text{admission}}$ has the same effect; however, $\delta_{\text{admission}}$ has lower impact than K in terms of the magnitude of change in delay bounds and maximum sustainable load. Moreover, since these parameters can be set independently, any operating point in the feasibility region for Harmony (shown in Figure 6) can be achieved.

6 Harmony benefits

Harmony enables bounded queueing and network delays, and eliminates congestion-related drops for each message. This leads to some immediate benefits. For instance, Harmony has the potential to enable efficient RDMA over converged Ethernet deployments. Specifically, most current RDMA deployments use Priority Flow Control (PFC), a mechanism to enable lossless network fabrics (no buffer overflows and packet drops within the network fabric) [1, 44, 49, 72]. While PFC enables lossless network fabrics, it suffers from several undesirable problems: head-of-line blocking, inevitable deadlocks, congestion spreading, reduced network utilization and increased tail latencies [34–36, 44, 50, 72]. Harmony, by design, eliminates congestion-related drops thus enabling lossless networks without the need for PFC.

However, to truly realize Harmony benefits, Harmony must be extended to provide end-to-end delay guarantees. Specifically, we need extensions in Harmony to bound host-side processing delays (*e.g.*, due to slow software [20]), and to



(a) Varying RPC sizes (b) Varying Oversubscription (c) Varying Topology Types

Figure 8: **Sensitivity of Harmony performance with (a) varying RPC sizes (b) varying oversubscription ratio in network topologies; and (c) varying scale of topologies (two-tier versus three-tier)**. Discussion in §5.2.

eliminate packet queueing and delays at the host (*e.g.*, due to host congestion [7,31]). To bound host-side processing delays, we observe that Harmony maintains the invariant of bounded queueing at switches thus allowing to move congestion control out of the fast path; as a result, Harmony can be easily integrated with existing high-performance userspace network stacks [38,46], in-kernel network stacks [21], and accelerator based network stacks [2,26,58]. These stacks often use dedicated cores and/or specialized hardware for packet processing, reducing host processing overheads to a tiny fraction of the network RTT. To eliminate queueing and packet drops at the host due to host congestion, Harmony can be easily integrated with recent mechanisms for host congestion control [7]. Finally, to extend Harmony's bounded delay guarantees all the way to the application layer, Harmony can be integrated with μs-scale network, CPU and storage schedulers [21,37,53]. We leave these extensions to the future; however, we demonstrate several applications that may benefit from such extensions.

To demonstrate potential application-layer benefits of Harmony, we use the DPDK-based prototype implementation of Harmony from §4, and use three dedicated cores—one for pacing data, one for control packet processing, and one for implementing the remainder of the logic—to minimize host processing delays. We integrate this implementation with several applications, while ensuring that network layer processing is isolated and host does not observe congestion. In extremely rare scenarios where Harmony incurs worst-case network-layer delays as well as host processing delays, applications may get a `reject` message—this does not violate any correctness guarantees, and simply requires applications to retry; nevertheless, to account for these delays, we simply add 1μs to Harmony's end-to-end delay bound. This prototype implementation thus offers bounded message delays between sender-side and receiver-side Harmony interfaces. We integrate it with three applications, that we discuss next.

CPU-efficient storage stacks. Disaggregated storage has become common in today's datacenters. As a result, modern storage stacks have been integrated with network transports (e.g., NVMe-over-Fabrics) in order to facilitate access to remote storage devices. Today's storage stacks rely on one of the two designs: polling-based or interrupt-based mechanisms. Polling-based designs provide extremely good latency when applications are run in isolation, but suffer when applications share CPU resources. Interrupt-based designs work better in the shared scenario, however have suboptimal CPU-efficiency due to frequent context-switches. Harmony's predictable delay guarantees enable a new point in the design space of CPU-efficient storage stacks: improved CPU efficiency without sacrificing tail latency. Specifically, storage stack running atop Harmony can use predictable network delays in Harmony to avoid both interrupts or polling—applications can be scheduled more precisely using the delay bound. In [5], we demonstrate the feasibility using the Harmony prototype: we find that Harmony enables new operating points in terms of tail latency and system throughput.

Efficient host packet processing pipelines. Recent work [20, 28] has demonstrated that CPU-efficiency of host packet processing pipelines can degrade significantly when multiple applications/connections running on a single CPU core contend for cache capacity, leading to higher cache misses. Recent work on Reframer [28] tries to reduce these misses by deliberately buffering packets, and waiting for a fixed amount of time for a batch worth of packets to arrive per application, before allowing them to be processed by the CPU. The choice of timeout value presents an inherent tradeoff: a larger timeout value results in better CPU-efficiency due to more packets getting batched at the cost of higher latency, while a smaller timeout value results in lower latency but worse CPU-efficiency. Determining the ideal timeout value is difficult in best-effort networks due to unpredictable network delays. Harmony's predictable network delay guarantees enable overcoming the above tradeoff: in absence of failures, receivers receive data at a fixed bandwidth, allowing to set a timeout to reap the maximum benefits of Reframer. Evaluation of our prototype, integrated with Reframer, achieves 2$\times$ higher throughput than the original Reframer [5].

Efficient failure detectors. Detecting host failures is a fundamental problem in distributed systems. Designing a failure detector that is reliable (i.e. provides small false positive probability) and fast (provides failure notification within a small delay) is a hard problem, especially in best-effort networks where network delays are unpredictable [43]. Harmony, using its bounded delay guarantees, has the potential to allow

achieving both these goals simultaneously. In [5], we present a prototype failure detection built on top of Harmony. The failure detector probes the target host by issuing a `request` and waiting for an `rsvp`. Given Harmony's predictable delay guarantees, the failure detector knows exactly when to expect the corresponding `rsvp` message. The absence of such a message implies there was a failure, either at the host or in network hardware. We show analytically that, for modern datacenter networks, the probability of the probe encountering a network failure is relatively small (and decreases with the number of tries) due to large path diversity; based on this, we are able to demonstrate the feasibility of designing low-latency host failure detectors that also have low false positive rates.

7 Related Work

Our exploration of congestion-free datacenter architecture is related to three key areas of research.

Pre-datacenter network designs (RSVP, virtual circuit switching, ATM networks, etc.). There have been several attempts to designing Internet architectures with predictable performance, *e.g.*, using Resource ReSerVation protocol (RSVP) [70, 71], virtual circuit switching [12, 40, 56], and hop-by-hop flow control in ATM networks [17, 54], to name a few. Realizing predictable performance on the Internet faced several challenges: large RTTs, the lack of a single administrative entity precluding support from hosts and routers, and potential deadlocks due to policy-driven routing [17, 54]. These challenges proved to be insurmountable in the Internet context; however, the equation is quite different for modern datacenter networks: they support small RTTs, operate within a single administrative domain allowing us to leverage both host and switch support, and are already exploring programmable switches and custom-designed network interface cards (NICs) with more powerful interfaces than commodity hardware. Our work builds upon decades of work on predictable Internet architectures, but advances them significantly: combining the idea of virtual channels with RSVP to avoid throughput loss in datacenter networks and presenting analytical bounds on bounded queueing in the datacenter context.

Circuit-switched networks. Circuit-switched datacenter network architectures [10, 25, 42, 48, 57, 63], by establishing an end-to-end dedicated path prior to data transmission, enable bounded queueing at each switch in the network. Our goals are aligned with those in circuit-switched networks; unsurprisingly, some of our ideas resemble the techniques used in circuit-switched networks, *e.g.*, wavelength-switching [14, 16, 52] and packet-based optical switching [24, 30]. However, there are two—fundamental—differences. First, our work demonstrates that it is possible to achieve powerful guarantees using distributed packet-switched networks, the primary deployment scenario in today's datacenter networks; recent results [10] demonstrate that it is impossible to simultaneously achieve low latency and high network utilization for all workloads using distributed circuit-switched networks. Second, unlike circuit-switched networks that assume host and network hardware clocks to be perfectly synchronized (that is known to be hard at the datacenter scale [69]), our design does not make any assumptions on clock synchronization.

Predictable and low-latency datacenter network designs. Most of the existing datacenter network designs focus on best-effort delivery [6, 8, 9, 19, 27, 29, 32, 33, 44, 51]. There are two exceptions. The first exception is the recent work on lossless network designs [1, 44, 72]; while these techniques ensure that packets are never dropped due to buffer overflow, by design, they can suffer from packet stalls—packets can be queued in switch buffers for an unpredictable amount of time due to PFC pause frames [1, 44]. Thus, they do not guarantee bounded network delays. Harmony enables lossless networks by design (in failure-free scenarios), thus offering a way to realize RDMA over converged Ethernet without PFC.

Another closely related line of work [15, 39, 55, 69] is on predictable network performance using centralized schedulers. These designs suffer from the usual centralized design limitations, namely scalability and availability, especially for high-bandwidth links. Harmony focuses on achieving predictable network performance over distributed packet-switched networks. Indeed, our evaluation results suggest that Harmony achieves performance similar to centralized designs while enjoying the benefits of a distributed datacenter architecture.

8 Conclusion

Existing datacenter networks provide best-effort delivery, that is, messages may experience unpredictable queueing, delays, and congestion-related drops due to switch buffer overflows within the network. System designers are thus forced to rely on minimal assumptions from the network, resulting in inefficiency and complexity in host hardware and software. This paper argues for datacenter architectures that provide stronger guarantees by design. We have presented Harmony, a datacenter architecture that provides congestion-free message delivery guarantees—each message experiences bounded queueing at each switch in the network. Harmony thus ensures that network delays are bounded in failure-free scenarios, and that congestion-related drops are eliminated. We establish that Harmony provides these properties with near-zero overheads compared to best-effort delivery networks.

Acknowledgements

We would like to thank our shepherd, Ravi Soundararajan, and NSDI reviewers for insightful feedback. We would also like to thank Midhul Vuppalapati for many useful discussions during this project. This research was in part supported by NSF grants CNS-2047283 and a Sloan fellowship.

References

[1] 802.1Qbb – Priority-based Flow Control . https://1.ieee802.org/dcb/802-1qbb/.

[2] Annapurna Labs. http://www.annapurnalabs.com.

[3] Portable Switch Architecture (PSA) . https://p4.org/p4-spec/docs/PSA.html.

[4] Switch Buffer Size. https://people.ucsc.edu/~warner/buffer.html.

[5] Harmony Technical Report. https://github.com/communication-harmony/tech-report.

[6] V. Addanki, O. Michel, and S. Schmid. PowerTCP: Pushing the performance limits of datacenter networks. In *USENIX NSDI*, 2022.

[7] S. Agarwal, A. Krishnamurthy, and R. Agarwal. Host congestion control. In *ACM SIGCOMM*, 2023.

[8] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data Center TCP (DCTCP). In *ACM SIGCOMM*, 2011.

[9] M. Alizadeh, S. Yang, M. Sharif, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker. pFabric: Minimal Near-optimal Datacenter Transport. In *ACM SIGCOMM*, 2013.

[10] D. Amir, T. Wilson, V. Shrivastav, H. Weatherspoon, R. Kleinberg, and R. Agarwal. Optimal oblivious reconfigurable networks. In *ACM STOC*, 2022.

[11] S. Arslan, Y. Li, G. Kumar, and N. Dukkipati. Bolt: Sub-rtt congestion control for ultra-low latency. In *USENIX NSDI*, 2023.

[12] J. Aspnes, Y. Azar, A. Fiat, S. Plotkin, and O. Waarts. On-line Routing of Virtual Circuits with Applications to Load Balancing and Machine Scheduling. In *JACM*, 1997.

[13] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload analysis of a large-scale key-value store. In *ACM SIGMETRICS*, 2012.

[14] I. Baldine, G. N. Rouskas, H. G. Perros, and D. Stevenson. JumpStart: A Just-in-time Signaling Architecture for WDM Burst-switched Networks. In *IEEE communications magazine*, 2002.

[15] H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron. Towards Predictable Datacenter Networks. In *ACM SIGCOMM*, 2011.

[16] D. Banerjee and B. Mukherjee. Wavelength-routed Optical Networks: Linear Formulation, Resource Budgeting Tradeoffs, and a Reconfiguration Study. In *IEEE/ACM ToN*, 2000.

[17] C. Basso, J. Calvignac, D. Orsatti, and F. Verplanken. Hop-by-hop Flow Control in an ATM Network, 1998. US Patent.

[18] T. Benson, A. Akella, and D. A. Maltz. Network traffic characteristics of data centers in the wild. In *ACM IMC*, 2010.

[19] Q. Cai, M. T. Arashloo, and R. Agarwal. dcPIM: Near-optimal Proactive Datacenter Transport. In *ACM SIGCOMM*, 2022.

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

[21] Q. Cai, M. Vuppalapati, J. Hwang, C. Kozyrakis, and R. Agarwal. Towards μs Tail Latency and Terabit Ethernet: Disaggregating the Host Network Stack. In *ACM SIGCOMM*, 2022.

[22] J. G. Dai and B. Prabhakar. The throughput of data switches with and without speedup. In *IEEE INFOCOM*, 2000.

[23] W. J. Dally, P. P. Carvey, L. R. Dennison, and P. A. King. Router with Virtual Channel Allocation, 2003. US Patent.

[24] T. S. El-Bawab and J.-D. Shin. Optical Packet Switching in Core Networks: between Vision and Reality. In *IEEE Communications Magazine*, 2002.

[25] N. Farrington, G. Porter, S. Radhakrishnan, H. H. Bazzaz, V. Subramanya, Y. Fainman, G. Papen, and A. Vahdat. Helios: a Hybrid Electrical/optical Switch Architecture for Modular Data Centers. In *ACM SIGCOMM*, 2010.

[26] D. Firestone, A. Putnam, S. Mundkur, D. Chiou, A. Dabagh, M. Andrewartha, H. Angepat, V. Bhanu, A. Caulfield, E. Chung, et al. Azure Accelerated Networking: SmartNICs in the Public Cloud. In *USENIX NSDI*, 2018.

[27] P. X. Gao, A. Narayan, G. Kumar, R. Agarwal, S. Ratnasamy, and S. Shenker. pHost: Distributed Near-optimal Datacenter Transport Over Commodity Network Fabric. In *ACM CoNEXT*, 2015.

[28] H. Ghasemirahni, T. Barbette, G. Katsikas, A. Farshin, A. Girondi, Massimoand Roozbeh, M. Chiesa, G. Maguire, and D. Kostic. Packet Order Matters! Improving Application Performance by Deliberately Delaying Packets. In *USENIX NSDI*, 2022.

[29] M. P. Grosvenor, M. Schwarzkopf, I. Gog, R. N. Watson, A. W. Moore, S. Hand, and J. Crowcroft. Queues Don't Matter When You Can Jump Them! In *USENIX NSDI*, 2015.

[30] C. Guillemot, M. Renaud, P. Gambini, C. Janz, I. Andonovic, R. Bauknecht, B. Bostica, M. Burzio, F. Callegati, M. Casoni, et al. Transparent Optical Packet Switching: The European ACTS KEOPS Project Approach. In *Journal of lightwave technology*, 1998.

[31] S. Han, N. Egi, A. Panda, S. Ratnasamy, G. Shi, and S. Shenker. Network support for resource disaggregation in next-generation datacenters. In *ACM Workshop on Hot Topics in Networks (HotNets)*, 2013.

[32] M. Handley, C. Raiciu, A. Agache, A. Voinescu, A. W. Moore, G. Antichi, and M. Wójcik. Re-architecting Datacenter Networks and Stacks for Low Latency and High Performance. In *ACM SIGCOMM*, 2017.

[33] S. Hu, W. Bai, G. Zeng, Z. Wang, B. Qiao, K. Chen, K. Tan, and Y. Wang. Aeolus: a Building Block for Proactive Transport in Datacenters. In *ACM SIGCOMM*, 2020.

[34] S. Hu, Y. Zhu, P. Cheng, C. Guo, K. Tan, J. Padhye, and K. Chen. Deadlocks in datacenter networks: Why do they form, and how to avoid them. In *ACM HotNets*, 2016.

[35] S. Hu, Y. Zhu, P. Cheng, C. Guo, K. Tan, J. Padhye, and K. Chen. Deadlocks in datacenter networks: Why do they form, and how to avoid them. In *ACM HotNets*, 2016.

[36] S. Hu, Y. Zhu, P. Cheng, C. Guo, K. Tan, J. Padhye, and K. Chen. Tagger: Practical pfc deadlock prevention in data center networks. In *ACM CoNext*, 2017.

[37] J. Hwang, M. Vuppalapati, S. Peter, and R. Agarwal. Rearchitecting Linux Storage Stack for μs Latency and High Throughput. In *USENIX OSDI*, 2021.

[38] A. Kalia, M. Kaminsky, and D. Andersen. Datacenter RPCs can be General and Fast. In *USENIX NSDI*, 2019.

[39] R. Kapoor, G. Porter, M. Tewari, G. M. Voelker, and A. Vahdat. Chronos: Predictable Low Latency for Data Center Applications. In *ACM SoCC*, 2012.

[40] S. Keshav and S. Kesahv. *An Engineering Approach to Computer Networking: ATM networks, the Internet, and the Telephone Network*. Addison-Wesley Reading, 1997.

[41] G. Kumar, N. Dukkipati, K. Jang, H. M. Wassel, X. Wu, B. Montazeri, Y. Wang, K. Springborn, C. Alfeld, M. Ryan, et al. Swift: Delay is Simple and Effective for Congestion Control in the Datacenter. In *ACM SIGCOMM*, 2020.

[42] S. Legtchenko, N. Chen, D. Cletheroe, A. Rowstron, H. Williams, and X. Zhao. XFabric: a Reconfigurable In-rack Network for Rack-scale Computers. In *USENIX NSDI*, 2016.

[43] J. B. Leners, H. Wu, W.-L. Hung, M. K. Aguilera, and M. Walfish. Detecting Failures in Distributed Systems with the Falcon Spy Network. In *ACM SOSP*, 2011.

[44] Y. Li, R. Miao, H. H. Liu, Y. Zhuang, F. Feng, L. Tang, Z. Cao, M. Zhang, F. Kelly, M. Alizadeh, et al. HPCC: High Precision Congestion Control. In *SIGCOMM*. 2019.

[45] M. A. Marsan, E. Leonardi, M. Mellia, and F. Neri. On the stability of input-buffer cell switches with speed-up. In *IEEE INFOCOM*, 2000.

[46] M. Marty, M. de Kruijf, J. Adriaens, C. Alfeld, S. Bauer, C. Contavalli, M. Dalton, N. Dukkipati, W. C. Evans, S. Gribble, et al. Snap: a Microkernel Approach to Host Networking. In *ACM SOSP*, 2019.

[47] N. McKeown, A. Mekkittikul, V. Anantharam, and J. Walrand. Achieving 100% throughput in an input-queued switch. *IEEE Transactions on Communications*, 1999.

[48] W. M. Mellette, R. McGuinness, A. Roy, A. Forencich, G. Papen, A. C. Snoeren, and G. Porter. Rotornet: A Scalable, Low-complexity, Optical Datacenter Network. In *ACM SIGCOMM*, 2017.

[49] R. Mittal, N. Dukkipati, E. Blem, H. Wassel, M. Ghobadi, A. Vahdat, Y. Wang, D. Wetherall, D. Zats, et al. TIMELY: RTT-based Congestion Control for the Datacenter. In *ACM SIGCOMM*, 2015.

[50] R. Mittal, A. Shpiner, A. Panda, E. Zahavi, A. Krishnamurthy, S. Ratnasamy, and S. Shenker. Revisiting Network Support for RDMA. In *ACM SIGCOMM*, 2018.

[51] B. Montazeri, Y. Li, M. Alizadeh, and J. Ousterhout. Homa: A Receiver-Driven Low-Latency Transport Protocol Using Network Priorities. In *ACM SIGCOMM*, 2018.

[52] C. S. R. Murthy and M. Gurusamy. *WDM Optical Networks: Concepts, Design, and Algorithms*. Prentice Hall, 2002.

[53] A. Ousterhout, J. Fried, J. Behrens, A. Belay, and H. Balakrishnan. Shenango: Achieving High {CPU} Efficiency for Latency-sensitive Datacenter Workloads. In *USENIX NSDI*, 2019.

[54] C. Özveren, R. Simcoe, and G. Varghese. Reliable and Efficient Hop-by-hop Flow Control. In *ACM SIGCOMM*, 1994.

[55] J. Perry, A. Ousterhout, H. Balakrishnan, D. Shah, and H. Fugal. Fastpass: A Centralized "Zero-Queue" Datacenter Network. In *ACM SIGCOMM*, 2014.

[56] S. Plotkin. Competitive Routing of Virtual Circuits in ATM Networks. In *IEEE JSAC*, 1995.

[57] G. Porter, R. Strong, N. Farrington, A. Forencich, P. Chen-Sun, T. Rosing, Y. Fainman, G. Papen, and A. Vahdat. Integrating Microsecond Circuit Switching into the Data Center. In *ACM SIGCOMM CCR*, 2013.

[58] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, et al. A Reconfigurable Fabric for Accelerating Large-scale Datacenter Services. In *IEEE/ACM ISCA*, 2014.

[59] M. Raab and A. Steger. "Balls into Bins"—A Simple and Tight Analysis. In *RANDOM*, 1998.

[60] J. Rexford. Tailoring Router Architectures to Performance Requirements in Cut-through Networks. 1999.

[61] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren. Inside the social network's (datacenter) network. In *SIGCOMM*, 2015.

[62] D. Shah. Maximal matching scheduling is good enough. In *IEEE GLOBECOM*, 2003.

[63] V. Shrivastav, A. Valadarsky, H. Ballani, P. Costa, K. S. Lee, H. Wang, R. Agarwal, and H. Weatherspoon. Shoal: A Network Architecture for Disaggregated Racks. In *USENIX NSDI*, 2019.

[64] A. Singh, J. Ong, A. Agarwal, G. Anderson, A. Armistead, R. Bannon, S. Boving, G. Desai, B. Felderman, P. Germano, and et al. Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter Network. In *ACM SIGCOMM*, 2015.

[65] R. Sivaram. Some measured google flow sizes. Technical report, 2008.

[66] P. Stuedi, A. Trivedi, B. Metzler, and J. Pfefferle. DaRPC: Data Center RPC. In *ACM SoCC*, 2014.

[67] P. Tammana, R. Agarwal, and M. Lee. Cherrypick: Tracing Packet Trajectory in Software-defined Datacenter Networks. In *ACM SOSR*, 2015.

[68] P. Tammana, R. Agarwal, and M. Lee. Simplifying Network Debugging with PathDump. In *USENIX OSDI*, 2016.

[69] B. C. Vattikonda, G. Porter, A. Vahdat, and A. C. Snoeren. Practical TDMA for Datacenter Ethernet. In *ACM EuroSys*, 2012.

[70] L. Zhang, S. Berson, S. Herzog, S. Jamin, and R. Braden. RFC2205: Resource ReSerVation Protocol (RSVP) – Version 1 Functional Specification, 1997.

[71] L. Zhang, S. Deering, D. Estrin, S. Shenker, and D. Zappala. RSVP: A New Resource Reservation Protocol. 1993.

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