Building a Partial Communication Synchrony Abstraction on Asynchronous Datacenters: Fundamental Limits, Mechanisms, and Applications

Saksham Agarwal *Cornell University*
Qizhe Cai *Cornell University*
Rachit Agarwal *Cornell University*
David Shmoys *Cornell University*
Amin Vahdat *Google*

Abstract

Existing datacenter networks are asynchronous—it is hard, or even impossible, to bound the time taken by a message to be transmitted from the sender to the receiver. Such network asynchrony reduces the set of assumptions that system designers can rely upon from the underlying network, thus introducing inefficiency and complexity in end-host systems.

We present cosy, a datacenter network-layer abstraction that, without requiring clock synchronization at network and host hardware, enables partial communication synchrony—a message submitted to the sender-side cosy interface is delivered to the receiver-side cosy interface within a bounded amount of time.

Enabling a partial communication synchrony abstraction over asynchronous hardware has some overheads. We present an analytical characterization of the fundamental limits (in terms of message delays and network utilization), a distributed protocol that theoretically achieves performance close to the fundamental limit, and an implementation that closely matches theoretical performance bounds over a distributed testbed. We also demonstrate benefits of the cosy abstraction using three applications: a storage stack that simultaneously support μs-scale latency and high CPU utilization, a CPU-efficient host network stack, and a μs-scale failure detector.

1 Introduction

Datacenter networks today offer only *asynchronous* communication—it is hard, or even impossible, to bound the time taken by a message to be transmitted from the sender to the receiver. For instance, recent deployment studies report that even state-of-the-art networks can introduce unpredictable message delays that can vary by three orders of magnitude [60]. Such unpredictability and variability in message delays reduces the set of assumptions that system designers can rely upon from the underlying network. Thus, to operate correctly on top of such asynchronous networks, distributed systems must embrace inefficiency and complexity. For instance, unpredictable and variable message delays enforce systems to choose between interrupts (high tail latencies) or continuously spin-polling the network interface (high CPU wastage) [19, 45, 48, 51, 65, 68]. As another concrete example, unpredictable and variable message delays result in poor spatial and temporal cache locality in network stacks, significantly impacting CPU efficiency for high-speed networks [20, 31, 32, 38, 79]. A now-long line of work in practice and theory of distributed computing has established the many additional limitations of networks supporting only asynchronous communication [24, 27, 28, 56, 71].

This paper presents `cosy`, a datacenter network abstraction that enables partial communication synchrony—we provide a complete definition in §2, but at a high level, this abstraction enables a message submitted to the sender-side `cosy` interface at time t to be delivered to the receiver-side `cosy` interface within time $t+\delta$, where the *delay bound* δ is known at the time of the message submission. Pragmatically, we want a small δ, which includes three components: (i) message waiting time at the sender; (ii) host-side network layer processing and DMA time at both the sender and the receiver; and, (iii) message queueing, transmission and propagation times at network switches and links. Similar to many other systems, the fundamental limits on δ depend on the load (informally defined as the amount of traffic entering the network per unit time): beyond a certain *maximum sustainable* load Θ, one of the above three components must grow unboundedly making δ bound unachievable. This leads to our core technical goal: designing mechanisms that enable the `cosy` abstraction while minimizing the delay bound δ (that includes all of the three components above) and maximizing the sustainable load Θ.

As a first step towards realizing the above goal, this paper makes three core contributions. First, we analytically characterize best possible bounds on δ and Θ for *any* mechanism that enables the `cosy` abstraction. We also show that several possible datacenter network architectures—those employing circuit-switched technologies [14, 16, 30, 59, 62, 70], those assuming clock synchronization [67], and/or those employing centralized schedulers that orchestrate transfer of individual messages [15, 67, 80]—can enable the cosy abstraction with minimal modifications while achieving optimal δ and Θ. While interesting, the rest of the paper focuses on existing datacenter network architectures: these are distributed, employ packet-switched technologies, and do not assume fine-grained clock synchronization at host and network hardware.

Our second contribution is to enable the cosy abstraction that achieves near-optimal δ and Θ over distributed, packet-switched, datacenter networks without requiring clock synchronization at network and host hardware. To give a high-level overview of the cosy design, consider the three factors discussed above that contribute to δ. To minimize message queueing at network switches, cosy design carefully orchestrates network resources among competing messages by placing its intellectual roots in the classical Resource ReSerVation protocol (RSVP) [83, 84]: we show that by using RSVP on datacenter networks, it is possible to enable the cosy abstraction with optimal δ at low loads. However, even at moderate loads, this basic design can lead to high message waiting time at the sender. To minimize the message waiting time at the sender at high loads, cosy combines the idea of virtual channels in multicomputer networks [25, 26, 73] with RSVP to maintain the invariant that each switch observes bounded, albeit non-zero, queueing (thus slightly increasing δ compared to the low load case). We demonstrate—analytically and empirically—that allowing a small amount of queueing at the switches significantly increases maximum sustainable load Θ, and reduces the message waiting at the sender to a bare minimum as long as the load is less than the maximum sustainable load. Finally, to minimize host processing delays, we observe that cosy design maintains the invariant of bounded queueing at switches thus requiring no congestion control at the hosts; as a result, cosy can be easily integrated with kernel-bypass techniques [58, 85], end-host accelerator based network stacks [2, 3, 33, 72] and/or μs-scale schedulers [34, 65] to achieve small bounded host processing delays.

Our third contribution is an end-to-end implementation of the cosy abstraction using readily available programmable switches and DPDK-based hosts. The current cosy abstraction offers bounded message delays between sender-side cosy interface and receiver-side cosy interface; while this can potentially be extended to bounded application-layer delays (*e.g.*, by integrating the cosy abstraction with μs-scale schedulers like Shenango [65] and Caladan [34]), we demonstrate that the current cosy abstraction already provides benefits for several applications: (i) a SPDK-based storage stack that exploits the predictable message delays enabled by the cosy abstraction to simultaneously achieve μs-scale latency and high CPU utilization; (ii) a host network stack for high-speed networks that exploits the predictable message delays enabled by the cosy abstraction to achieve high spatial and temporal cache locality (and thus, high CPU efficiency); and (iii) a μs-scale host failure detector.

Our work demonstrates that datacenter networks offering the cosy abstraction can have powerful implications for future systems and networking infrastructure. There are two important caveats, however. First, our point is not that every application will benefit from the cosy abstraction—instead, we believe that datacenter networks should simultaneously support both the cosy abstraction and the classical best-effort delivery abstraction; applications running atop can choose which of the two interfaces they want to use depending on desirable performance goals (similar to today's network stacks that allow applications to choose between reliable and unreliable interfaces). Second, while we demonstrate the potential benefits of the cosy abstraction using several applications, we have only scratched the surface—as eloquently argued in [74, 81], there are many additional benefits of (partial) communication synchrony in datacenter networks; however, additional work will be needed at each layer of systems stack to reap end-to-end benefits of the cosy abstraction. While it may take longer than the lifetime of a single project to realize systems that efficiently exploit all the benefits of partial communication synchrony, we believe the potential benefits make it a worthwhile exploration.

2 The cosy abstraction

We begin this section by providing an overview of the cosy abstraction: the interface it offers to applications, a formal definition of the partial communication synchrony property it enables, and the core technical problem statement it solves (§2.1). We then explore the design space for enabling the cosy abstraction (§2.2). Recall that we target distributed, packet-switched, datacenter networks with fixed (shared) buffer size at each switch [8, 40, 77]. We make no assumptions on clock synchronization at host and network hardware.

2.1 Interface, definition & problem statement

The cosy application-network interface. Applications that want partial communication synchrony interact with cosy using a special cosy-RPC interface, implemented using standard submission/completion queues [49, 53, 58, 78]. Similar to most existing RPC interfaces, applications submit their messages using a submission queue at the sender-side cosy interface and the completion queue notifies the application of the message delivery status (using a `complete` or `failure` flag). The only difference in our interface is that the completion queue has one additional flag: `reject`, which indicates that cosy is unable to provide partial communication synchrony for the message at the time of message submission. If the flag is either `reject` or `failure`, the application can either resubmit the message using the cosy interface, or send it using existing best-delivery interface.

Underneath, the host-side cosy implementation uses standard kernel-bypass techniques (thus avoiding unpredictable kernel latencies); the current implementation runs on DPDK, and can be easily extended to user-space stacks [49, 58, 65], or to hardware implementations [11]. We provide details on end-to-end `cosy` implementation in §5.

Partial communication synchrony. The `cosy` abstraction enables partial communication synchrony, a network-layer property defined as follows. Let m be a message of a fixed size, and let a_m and r_m denote the time (according to the local clock of the sender) at which m was submitted to the submission queue, and the time at which the flag for m was updated at the completion queue. The partial communication synchrony property enables by the `cosy` abstraction ensures that $r_m - a_m \leq \delta$, where δ is an absolute constant that depends only on network hardware (thus, is known at the time of the message submission). Note that δ includes all of the three delay components discussed in §1: (i) message waiting time at the sender; (ii) host-side network layer processing and DMA time at both the sender and the receiver; and, (iii) message queueing, transmission and propagation times at network switches and links.

Problem Statement. Pragmatically, we want to design the `cosy` abstraction that achieves a small δ. As mentioned earlier, the achievable values of δ depend on the network load defined as the ratio of the total amount of traffic generated by applications at the senders in any time window of length T, and the total access link bandwidth across all senders. We define the maximum sustainable load Θ as the maximum network load for which `cosy` provides partial communication synchrony for each message (that is, no message is rejected). As we will demonstrate, there is a tight relationship between δ and Θ: intuitively, as the desired maximum sustainable load increases (no messages must be rejected despite higher network loads), it becomes necessary to have higher values of δ. This leads to our problem statement: we want to design the `cosy` abstraction that minimizes the delay bound δ and simultaneously maximizes the maximum sustainable load Θ.

2.2 Design Space

We now discuss existing mechanisms that can be used to enable partial communication synchrony, and outline how `cosy` builds upon these mechanisms.

Pre-datacenter packet-switched 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) [83, 84], virtual circuit switching [12, 52, 69], and hop-by-hop flow control in ATM networks [18, 66], to name a few. Realizing these architectures on the Internet faced several challenges: large round trip times (RTT), the lack of a single administrative entity precluding support from end hosts and network routers, and potential deadlocks due to policy-driven routing [18, 66]. In addition, designs from ATM networks are not compatible with IP/Ethernet, the typical deployment scenario in datacenter networks. These challenges proved to be insurmountable in the Internet context; however, the equation is quite different for modern datacenter networks: network hardware can support single-digit microsecond RTTs, these networks operate within a single administrative domain allowing us to leverage both end host and switch support and most datacenter providers are already exploring programmable switches and custom-designed network interface cards (NICs) [8, 57, 60, 77, 85] with more powerful interfaces than commodity hardware. Our work builds upon decades of work on predictable Internet architectures, but advances them significantly: incorporating techniques to optimize maximum sustainable throughput in datacenter networks, presenting the first analytical bounds in the datacenter context, and demonstrating the usefulness of the `cosy` abstraction using multiple datacenter applications (§1, §3).

Circuit-switched network designs. There has been decades of work on designing circuit-switched networks, including recent work that focuses on datacenter networks [14, 16, 30, 39, 54, 59, 62, 70, 76] and this list barely scratches the surface of work from pre-datacenter era. These networks, by establishing an end-to-end dedicated path prior to data transmission, enable communication synchrony. 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, 62] and packet-based optical switching [29, 42]. However, there are two—fundamental—differences. First, unlike circuit-switched networks that assume host and network hardware clocks to be synchronized, we demonstrate that partial communication synchrony can be achieved even without clock synchronization (that is known to be hard at the datacenter scale [80]). Second, unlike circuit-switched network designs that necessitate zero buffering at each network switch, our work explores the benefits of bounded (but not necessarily zero) queueing at switches. As we will discuss, allowing even a small amount of queueing at network switches not only leads to substantially different designs but also higher sustainable loads when compared to idealized distributed circuit-switched networks.

Existing datacenter network designs. Most of the existing datacenter network designs focus on best-effort delivery [9, 10, 13, 22, 36, 41, 43, 44, 57, 61]. There are two exceptions. The first exception is the recent work on lossless network designs [1, 57, 85]; while these techniques ensure that packets are never dropped due to buffer overflow, by design, they can suffer from packet stalls where packets can be queued in switch buffers for an unpredictable amount of time due to PFC pause frames [1]; as a result, they do not enable partial communication synchrony as defined in previous subsection. Designing lossless network protocols that does not use PFC is indeed an open problem [57]. Another line of work [15, 50, 67, 80] is the most related to ours: they focus on predictable network-level performance using centralized schedulers. However, they suffer from the classical problems of centralized designs, namely scalability and availability (while they were able to operate on 10Gbps links, scaling them up for modern 100Gbps and higher link bandwidths is non-trivial due to centralized server becoming a bottleneck); moreover, many of these require clock synchronization that, as mentioned above, is hard to achieve at the datacenter scale [80]. Our work explores the problem of achieving partial communication synchrony over distributed packet-switched networks without clock synchronization.

3 Realizing the `cosy` abstraction

This section explores design of mechanisms for distributed packet-switched datacenter networks that enable the `cosy` abstraction. We first establish fundamental limits on the delay bound δ and maximum sustainable load Θ for *any* design that enables the `cosy` abstraction (§3.1). We then demonstrate that how to achieve optimal δ at low loads (§3.2). Finally, we present `cosy` design that enables a smooth tradeoff space between the delay bound and the maximum sustainable load (§3.3); we characterize this tradeoff space in §3.4.

3.1 Fundamental Limits

The following theorem outlines the fundamental limits on the delay bound ($\delta^\star$) and maximum sustainable load ($\Theta^\star$) possible for any network that enables the `cosy` properties.

Theorem 3.1 *Let hRTT denote the unloaded RTT of the network*[1]*, let B be the bandwidth for each link, and let s be the size of each message. Then, for any design that enables partial communication synchrony, the following bounds hold:*

$$\delta_{synchrony} \geq \delta^\star = 2 \cdot \text{hRTT} + \frac{s}{B}$$

$$\Theta_{synchrony} \leq \Theta^\star = \frac{1}{2(1+\alpha)},$$

where α is the ratio of hRTT and s/B. These bounds are tight.

We present the formal proof in Appendix A. Intuitively, our proof for the delay bound demonstrates a traffic matrix for which achieving partial communication synchrony is impossible without waiting at least one hRTT before transmitting each message; the remainder of the expression is simply the transmission time of the message at the end-host, the time it takes for the last packet in the message to traverse the network, and the time for the receiver to acknowledge completion of the message. Our proof for the bound on sustainable loads uses a folklore result on bipartite matchings to get a factor of $1/2$ (along with an equivalence between the matching size and network utilization for large-enough message sizes); the $(1+\alpha)$ factor captures the fact that, for small messages, there is some loss of sustainable load due to the overheads of the first hRTT. We demonstrate the tightness of the bounds by showing that it is possible to use existing centralized designs [15, 67, 80], with minimal modifications, along with clock synchronization at all host and network hardware to enable partial communication synchrony while achieving the bounds in Theorem 3.1.

We make two observations. First, simple calculations for state-of-the-art network hardware shows that, even for tiny messages, $\delta^\star$ is within $1.5\times$ of the best possible delay for asynchronous networks ($\delta^\star$ gets closer to best possible delay as message sizes increase); thus, if we can design mechanisms that achieve delay close to this lower bound, it would be possible to achieve partial communication synchrony with delay comparable to state-of-the-art best-effort transport protocols [43, 44, 53, 57, 61]. The second observation relates to the sustainable load bound—for small value of α (or, alternatively, large message sizes as in classical analytical results on network throughput [9, 47, 57]), the bound converges to the best possible folklore bound on sustainable loads for asynchronous networks. The rest of the paper focuses on exploring the design space for distributed packet-switched network designs that enable partial communication synchrony.

3.2 Achieving optimal δ for low loads

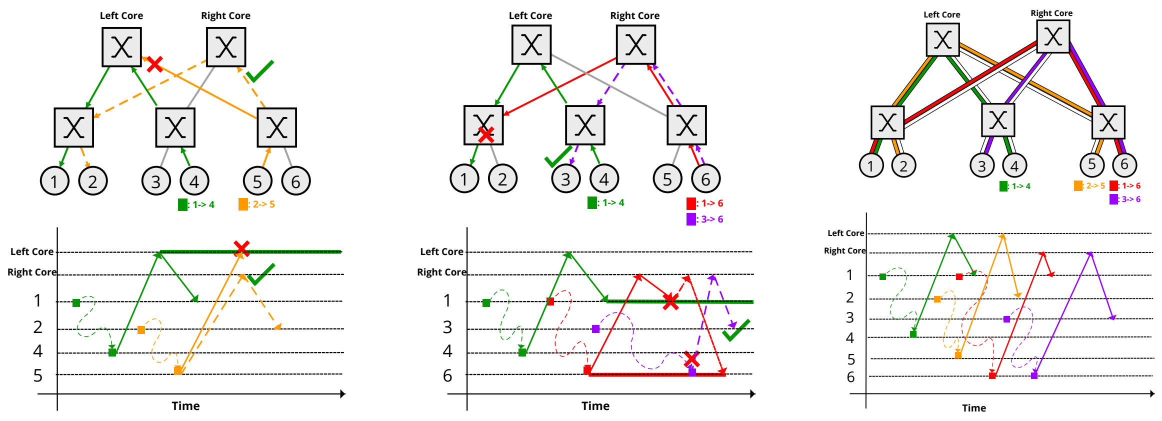
This subsection presents one design point for the `cosy` abstraction: one that achieves optimal δ (from Theorem 3.1), but only at low loads, by using the classical Resource ReSerVation protocol (RSVP). We show that, when put together with shortest multipath routing in a specific manner, this design enables partial communication synchrony with the optimal delay bound but with suboptimal maximum sustainable loads. The design works as follows:

- Each sender, upon a message arrival at its submission queue, sends a `request` control packet to the receiver. Upon receiving a `request`, if the receiver is not busy, it marks itself busy, and sends a `rsvp` to the corresponding sender. If the receiver is busy, it sends a `reject` to the sender indicating that partial communication synchrony cannot be guaranteed for the message due to high load.

- Each switch maintains, for each of its ports/links, whether the link is reserved or unreserved. Upon receiving a `rsvp`, a switch *uniform randomly* chooses one of the unreserved outgoing links along one of the shortest paths to the sender, embeds its identifier into the `rsvp` header[2], marks the link as reserved, and forwards the `rsvp` on to that link. If no unreserved 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).

[1]Defined as the round trip time for a single MTU-sized packet in the absence of any other packet in the network. This is a property of network hardware.

[2]The original RSVP protocol assumed an underlying routing algorithm, but MPLS-based realizations of RSVP as in commercially available routers are more general; we describe the protocol using routers embedding their identifiers in the header to make the discussion more concise.



(a) RSVP, Blocking in the network (b) Blocking at the end host (c) Reduced blocking with virtual links

Figure 1: **An illustration of the design in §3.2 (left), and its two problems: blocking in the network (left) and at the hosts (center). The example also demonstrates how the `cosy` design in §3.3 alleviates these problems using the idea of virtual links and end-host slots, for $K = 2$ (right). At the bottom of each figure is the timing diagram, demonstrating the sequence of events at each node (`requests` are shown in squiggly lines, `rsvps` and reserved links are shown in solid lines, and straight dotted lines represent `rsvp` that were never initiated). Discussion on this example in respective subsections.**

`reject` traverses the same set of switches that forwarded `rsvp` so that corresponding links can be unreserved. Importantly, only shortest paths are used to ensure deadlock freedom [18, 66].

- Each sender, upon receiving a `rsvp` either sends a `reject` (if busy sending a message to some other receiver) or, if idle, transfers the message at full rate using the switch identifiers in the `rsvp` header; once the message is finished, the sender sends a `complete` and marks itself free;
- Each switch, upon receiving the `reject` or `complete`, marks the corresponding link unreserved and forwards it toward the receiver;
- Each receiver, upon receiving `reject` or `complete`, marks itself idle; 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 in the completion queue for the message; if the sender does not receive a `reject` or `complete` before time $\delta = \delta^{\star}$, it marks its completion queue with a `reject` flag for the message (this handles packet drops due to failures).

Figure 1(left) illustrates the above design for an example. Intuitively, at low loads, this design provides partial communication synchrony because it satisfies two sufficient conditions: since each message is transmitted along a reserved path, the arrival rate for each message perfectly matches the outgoing bandwidth available for that message; as a result, there is no transient and/or persistent queueing at any switch, resulting in zero queueing delays and zero congestion-related drops. The following theorem summarizes the achievable performance for the above design:

Theorem 3.2 *For the above design, we have that $\delta = \delta^{\star}$, and that expected maximum sustainable load, randomized over choice of* `rsvp` *forwarding decisions, is given by $\mathbb{E}[\Theta_{RSVP}] = (1 - 1/e)\Theta^{\star}$, where $e \approx 2.72$ is base of the natural logarithm.*

At a high-level, our proof for Theorem 3.2 uses a novel connection between the classical balls-and-bins problem and the problem of computing expected maximum sustainable load for partial communication synchrony. The theorem, similar to previous analytical results [67], 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.

Understanding the root causes for low sustainable load. The above design resembles several prior designs, *e.g.* virtual circuit switching [12, 52, 69] and distributed circuit-switched networks [14, 16, 29, 59, 76], in that, each link is used by a single sender-receiver pair at any point of time. The low sustainable load problem is also same as faced in prior techniques: "blocking" effect, both inside the network and at the hosts. We discuss these below (Figure 1 shows an example).

Blocking in the network. In the above design, switches make decisions on forwarding `rsvp` based on "local" information, without any view of the state of links at neighboring switches. This could lead to requests being rejected even if there is a path available in the network. For instance, consider the example shown in Figure 1a: 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` is dropped. Had the leaf chosen the right spine switch, it would have successfully reached sender 2, improving sustainable load. This problem, caused by switches making uncoordinated decisions, is referred to as blocking in the network.

Blocking at the end host. The second blocking effect happens due to receivers making immediate decisions upon receiving `requests`, again based on "local" information. This leads to suboptimality in two scenarios: for the last `rsvp` sent by the receiver, it may receive a `reject` very soon if (1) there is no available path to the sender; or (2) sender is busy sending message to some other receiver. Since the receiver does not have information about the network and/or sender state, immediately rejecting a request may lead to suboptimal sustainable loads. An example is shown in Figure 1b: here, 6 first receives `request` for 1 (red) and sends an `rsvp` to 1, it will receive a `reject` since there is no unreserved path between 1 and 6. In the meantime another `request` from 3 (purple) arrives, but gets immediately rejected. If the `request` for 3 however was allowed to wait for a small amount of time before rejection, it could have been admitted since a path exists between 3 and 6. We call this blocking problem at the end host.

As we will soon show, such blocking at the network and end-hosts is the core reason for the significant gap of $\sim$37% between Θ_{RSVP} in Theorem 3.2 and $\Theta^{\star}$ in Theorem 3.1.

3.3 `cosy`: achieving near-optimal δ and Θ

We now present `cosy`, a design that achieves a smooth tradeoff between the delay bound and maximum sustainable load. The key insight in `cosy` design is that at the core of the two blocking problems in the previous subsection is the invariant maintained by the RSVP-based design—enforcing zero queueing at each network switch. Specifically, by transmitting each message on a dedicated pre-reserved path, the above design ensures that, at each switch, the arrival rate for each message perfectly matches the outgoing bandwidth available for that message. Thus, it enforces zero transient queueing *and* zero persistent queueing. However, to achieve partial communication synchrony, we only need the latter; that is, it is sufficient to achieve bounded (but not necessarily zero) transient queueing and zero persistent queueing at each switch. This section demonstrates that allowing even a small amount of transient queueing at network switches not only leads to substantially different network designs but also results in improved sustainable loads.

`cosy` ensures bounded transient queueing and zero persistent queueing at each switch by integrating the above RSVP-based design with another classical idea: virtual channels in multicomputer networks [25, 26, 73]. We first outline `cosy` design, followed by an intuitive description of how it achieves partial communication synchrony while enabling a tradeoff space between the delay bound and maximum sustainable load. We formally prove the bounds in the next subsection.

`cosy` core design. `cosy` extends RSVP-based design from the previous subsection along two directions[3]:

- **Virtual links:** Each physical link is logically decomposed into K virtual links; each virtual link now enables a sender to transmit a message using bandwidth B/K. Each virtual link can be reserved by at most one message (although a message may be allocated more than one virtual link).
- **End-host slots:** Each sender and receiver maintains K slots. Each receiver can send one `rsvp` per slot and receive one message per slot. Similarly, each sender can send one message per slot, albeit each message can be transmitted using bandwidth B/K. Each slot can be reserved by at most one message, although a message may be allocated more than one slots. Any sender-receiver pair may use as many slots to exchange messages, as long as they never send two messages using the same slot.

Figure 1c shows an example—when compared to Figure 1a and 1b, virtual links and end-host slots ensure that message $2 \to 5$ no longer experiences network blocking and message $1 \to 6$ no longer experiences end-host blocking since multiple messages can co-exist at individual physical links and end hosts. The above design, by naïvely using RSVP protocol over individual virtual links and slots, reserves a dedicated virtual path of bandwidth B/K for each message (we describe some optimizations in next section). Intuitively, the above design maintains two invariants. First, at any point of time, at most K messages use any link in the network; and second, for each link, the sum of transmission rates of all messages using the link is no more than the link bandwidth. The second invariant ensures zero persistent queueing since the rate at which packets 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 first invariant ensures that transient queueing is bounded since only a small fixed number of packets can arrive at the same time at any switch (in next subsection, we will bound the number in terms of B, K and maximum number of switches along any network path). Put together, these two invariants are sufficient for `cosy` to achieve communication synchrony.

The two techniques discussed above are not merely heuristics—they are necessary for `cosy` to achieve near-optimal delay bound and maximum sustainable load: the first technique alleviates the blocking problem in the network by enabling fine-grained sharing of network resources: since an `rsvp` message can be forwarded along any of the "virtual links", and there are a factor K more virtual links than physical links, it is not too hard to show that the probability of `rsvp` being forwarded on a path that leads to blocking in the network reduces significantly. The second technique, on the

[3]For brevity, we describe the design when all links have the same bandwidth; see Appendix A for extensions for non-uniform bandwidths.

other hand, help alleviate the blocking problem at the end hosts: receivers can now send multiple `rsvp` messages again increasing the probability of their choosing the sender that does not lead to blocking at the hosts. Put together, these techniques enable `cosy` to achieve near-optimality in terms of both the delay bound and maximum sustainable load.

We discuss, in §4, several additional design details for `cosy`, and several optimizations that `cosy` uses by exploiting its own bounded delay guarantees. We also discuss in §4 how `cosy` design can be integrated with existing datacenter transport designs to simultaneously support communication synchrony and best-effort delivery semantics. In what follows, we provide bounds on `cosy` performance in the next subsection.

3.4 `cosy` Tradeoff Space

In this section, we provide analytical results on the tradeoff space between latency bound and maximum sustainable load enabled by `cosy`. We start with a basic result:

Lemma 3.3 *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 bandwidth of the physical link, and p be the maximum packet size. Then, the maximum transient queueing at any switch is bounded by*

$$\hat{q} \leq \#H \cdot (K-1) \cdot p$$

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

$$\hat{\delta} \leq \frac{\#H \cdot (\#H+1) \cdot (K-1)}{2} \cdot \frac{p}{B}$$

Let $\hat{S}$ be the minimum buffer size across all switches, and let $K^\star = \hat{S}/(\#H \cdot p)$. Then, the first part of the lemma shows that for $K \leq K^\star$, `cosy` will ensure that transient queueing never grows beyond the switch buffer capacity. Thus, no packets will ever be dropped. For instance, for a million server datacenter organized around a FatTree topology using switches with 100Gbps link, 32MB buffer size [5] and with 1.5KB packet sizes, we get that: $K^\star \approx 4200$. The lemma not only guarantees that no packets will be dropped for $K \leq K^\star$, but also bounds the queueing delay seen by any packet: for the above parameters and for $K = 4$, we get that $\hat{\delta} = 5.4\mu$s. Thus, for $K \leq K^\star$, `cosy` design ensures communication synchrony. Next, the following theorem establishes the maximum sustainable load for `cosy`:

Theorem 3.4 *The expected maximum sustainable load for* `cosy` *is:*

$$\mathbb{E}[\Theta_{\texttt{cosy}}] = f(K,C) \cdot \Theta^\star \qquad (1)$$

where,

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

and C is the number of core switches in the topology.

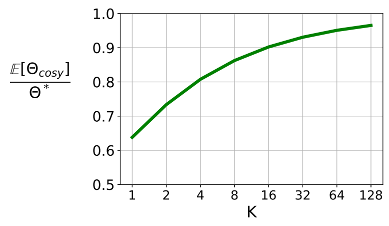


Figure 2: $\mathbb{E}[\Theta_{\texttt{cosy}}]$ normalized by $\Theta^\star$ from Eq. 1, for $C = 32$. Larger K enables larger sustainable load in expectation.

Figure 2 shows $\mathbb{E}[\Theta_{\texttt{cosy}}]$ values. Our results confirm the intuition discussed earlier: virtual links and end-host slots allow `cosy` to improve the maximum sustainable load, converging to $\Theta^\star$ (best possible maximum sustainable load) with larger K. Finally, we present the latency bound for `cosy`:

Theorem 3.5 *For loads less than the maximum sustainable load,* `cosy` *guarantees partial communication synchrony for*

$$\delta_{\texttt{cosy}} = 2 \cdot \text{hRTT} + \frac{s \cdot K}{B} + \hat{\delta}$$

The proofs for the theorems are in Appendix A. As a final note, `cosy` enables a smooth tradeoff between latency and sustainable load using (its only) design parameter—K. As K increases, the latency bound increases but `cosy` can also sustain much higher loads. As is usual, we will show that performance observed in empirical results (§5) is much better than the above bounds. Bridging the gap between `cosy` performance and the fundamental limits in Theorem 3.1, using distributed mechanisms, remains to be one of the most exciting problems in this direction.

4 `cosy` Design Details

We now provide some low-level details on `cosy` design, with a focus on how `cosy` exploits synchrony to enable fast failure reaction, achieve high utilization in practice, and efficiently handle best-effort delivery traffic.

Exploiting synchrony for fast failure reaction (including grey failures). `cosy` eliminates congestion-related drops. However, (inevitable) hardware failures and grey failures can still lead to dropping of inflight data packets. `cosy` enables fast failure reaction using the insight that all data packets in `cosy` traversing a bounded-queue path means that receivers know exactly when to expect data packets (or a `reject` message) corresponding to a `rsvp` message. Specifically, since `cosy` receivers know exactly when to expect the data packets (or a `reject` message) corresponding to a `rsvp` message, the absence of one of these two events allows the receiver to infer that a hardware failure has occurred. Thus, the receiver sends a `failure` control packet to the sender[4], and the sender could

[4] `failure` control packet is sent with the highest priority. Since there are few simultaneous hardware failures, it would not affect the bounds in practice.

react extremely quickly—within time hRTT+$\hat{\delta}$ ($\approx 11\mu$s for modern hardware).

Exploiting synchrony for consistent switch state under failures. In addition to the routing state already maintained at switches, `cosy` switches maintain a small constant amount of state: for each virtual link, the switches store whether or not the virtual link is reserved, and if reserved, which outgoing virtual link is it "mapped to" (that is, the outgoing virtual link on which the corresponding `rsvp` message was forwarded to). For a switch with p physical links and K virtual links per physical link, this requires $p \cdot K + p \cdot K \log_2(pK)$ bits. For a large 128 port switch with $K = 4$, for instance, this would require merely 640 bytes worth of state; most switches have memory in megabytes [4, 5].

Inevitable hardware failures can also lead to dropping of control packets, resulting in "blocked connections" problem as in circuit switching—some of the reserved virtual links may not be unreserved due to dropping of control packets. `cosy` switches also exploit communication synchrony to resolve the blocked connection problem. To allow quick failure detection, each `cosy` receiver embeds a random sequence number (similar to initial sequence number in classical congestion control protocols) into each `rsvp` message. Each switch stores per-port state that maps the message and the random sequence number in the `rsvp` message to the virtual link that is reserved for that message. Again, due to bounded queueing, the switch knows that in the absence of hardware failures, this virtual link must be unreserved at time $\delta_{\texttt{cosy}}$. If such an event does not happen, switches can detect failures and clean up the state for the virtual link. Since programmable switches today do not provide a dataplane counter, our implementation approximates it using the control plane.

5 `cosy` Implementation and Evaluation

We have done an end-to-end implementation of `cosy` prototype on commodity end-hosts (in Linux hosts with DPDK) and commodity programmable switches with P4 programmability. This section provides some of the most interesting implementation details (§5.1); we use this implementation for our evaluation in a small-scale testbed (§5.2). We also evaluate `cosy` performance over large-scale topologies using a packet-level simulator (§5.3).

5.1 `cosy` Implementation

`cosy` end-host implementation. Much of our end-host implementation uses existing modules from prior network designs—generating and responding to `rsvp` messages is similar to the grant mechanism in Homa [61] and NDP [43]; and, packet header logic for routing and forwarding based on switch identifiers is similar to NDP and source routing [64]. `cosy` design requires senders to transmit messages at a rate that is dependent on the number of slots allocated to that message. Such "rate limit" functionality at the sender is already implemented in almost all network stacks [36, 43, 61]; `cosy` can use any of the existing implementations to enforce rate limits for individual messages. Finally, if end-host hardware offload were desirable, all of the above protocols (and respective functionalities) are implemented in [11], that can be used for hardware offload of `cosy` end-host functionality. The only real new aspect for the end-host implementation is allocation and deallocation of slots, and at the receiver, selection of messages to send `rsvp` packet to using power-of-two choices. These are fairly straightforward. Overall, our prototype implementation at the end-host uses ~3107 lines of code and two dedicated cores—one for pacing data and control packets, and the other for implementing the remainder of the logic.

`cosy` switch implementation. `cosy` switch implementation uses commercially available programmable switches [6]. The data plane of these switches employs Portable Switch Architecture [6], 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 ports, `cosy` parser implementation extracts the header, identifies the packet type (`rsvp`, `reject`, `complete`, or data packet), and forwards the packet to the ingress pipeline for further processing. The ingress pipeline decides the egress port to which the packet will be forwarded, and then passes the packet to the traffic manager (which then forwards the packet to the desired egress port). `cosy`'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). Using these data structures, `cosy` can be realized using the pipeline shown in Figure 3.

5.2 `cosy` Testbed Evaluation

We evaluate an end-to-end implementation of `cosy` on a small-scale testbed. This section focuses on `cosy`'s network-layer performance; the next section demonstrates applications that benefit from `cosy`'s semantics.

Evaluation Setup. We use a testbed with 8 servers organized along a two-tier topology with 10Gbps links. Since our implementation requires programmable switches, we could not use CloudLab or other large-scale topologies for evaluation our implementation. We use remote procedure calls (RPC) traffic using Poisson arrival process and an all-to-all traffic pattern; each RPC is a fixed size message to be sent from its sender to a randomly chosen receiver. We evaluate for varying RPC sizes from 8KB to 512KB; and unless specified otherwise the baseline size used is 128KB.

We measure message latency using the standard *slowdown* metric [44, 57, 61], defined the ratio of the message completion time using the workload, and the message completion

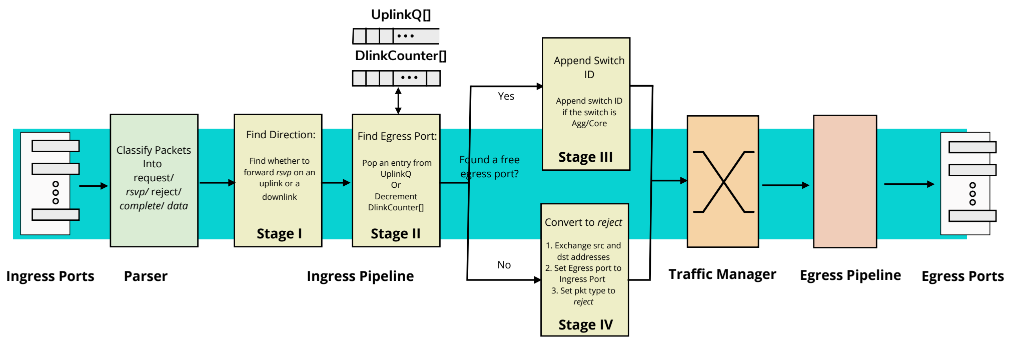
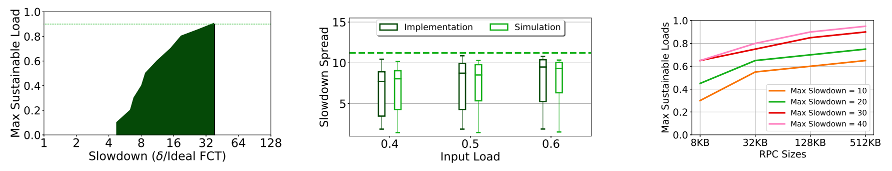

UplinkQ[]
DlinkCounter[]
Classify Packets Into request/ rsvp/ reject/ complete/ data
Find Direction:
Find whether to forward rsvp on an uplink or a downlink
Stage I
Find Egress Port:
Pop an entry from UplinkQ Or Decrement DlinkCounter[]
Stage II
Found a free egress port?
Yes
No
Append Switch ID
Append switch ID if the switch is Agg/Core
Stage III
Convert to reject
1. Exchange src and dst addresses
2. Set Egress port to Ingress Port
3. Set pkt type to reject
Stage IV
Ingress Ports
Parser
Ingress Pipeline
Traffic Manager
Egress Pipeline
Egress Ports

Figure 3: Implementation of cosy switch on programmable switches supporting PSA architecture. More description on §5.



(a) Max Sustainable Loads (b) Slowdown Spreads (c) Varying RPC Sizes

Figure 4: cosy evaluation on a 8-server testbed (a) cosy offers a tradeoff space between latency bound δ and maximum sustainable load—with larger latency bounds, cosy sustains higher loads (b), testbed results satisfy desired latency bounds and also near-perfectly match our simulation results (c), cosy sustains larger loads for increasing RPC sizes for fixed latency bounds

time in the absence of any other message in the network. For each evaluated workload, we present *slowdown spread*—{min, mean, 99%-ile, 99.9%-ile, and max} slowdowns using the boxes and whiskers—lower and upper whiskers represent min and max slowdown respectively; lower and upper box edges represent mean and 99.9%-ile and the mid-line in the boxes represents 99%-ile.

We now discuss the evaluation results shown in Figure 4.

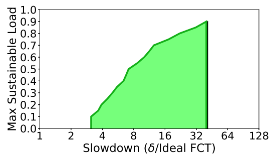
cosy enables communication synchrony while offering a unique tradeoff space between latency bound and sustainable load. The design parameter in cosy—number of virtual links (K)—allows it to offer a unique tradeoff space between latency bound and maximum sustainable load (shown as the shaded region in Figure 4(middle)). To characterize this space, we vary K from 1 to 16, and plot the pareto curve on the resulting latency bounds and sustainable loads. The tradeoff space enabled by cosy is intuitive and follows our analytical results—as the desired bound on communication synchrony latency is increased (via increasing K), cosy can sustain increasingly higher loads.

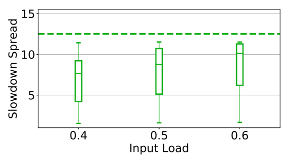
cosy empirical performance almost perfectly matches the analytical bounds. Over the evaluated workloads, we observe that cosy can sustain much higher loads than what the analytical bounds suggest. This is not surprising since, as discussed in §3.3, the analytical bound on maximum sustainable load assumes a worst-case traffic matrix. Our testbed results show that cosy is able to sustain loads as high as 0.9 for the baseline case of 128KB RPC size. Figure 4 shows that cosy also provides high values of maximum sustainable loads across wide range of RPC sizes. Although the worst-case theoretical bounds for message sizes smaller than hRTT could be quite low (Theorem 3.4), we observe in practice, even 8KB RPCs can sustain load of ~ 0.65 with maximum slowdown of 40.

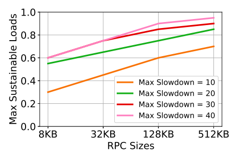
While our testbed results demonstrate the feasibility of cosy, more work is needed to evaluate cosy over large-scale deployments (we do not have large-scale testbeds with programmable switches). Nevertheless, we also implemented cosy on a packet-level simulator to evaluate its performance for large-scale datacenter topologies (discussed in next subsection). To verify the simulator robustness, 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 4. The testbed latency results near-perfectly match our simulator (which, in turn, matches the analytical bounds in Theorem 3.4). The slight difference in testbed and simulation results is due to two reasons: (1) our simulator uses a fixed PCIe latency, while the testbed exhibits some variance (usually a small value independent of message sizes [63]); and, (2) the timers used for rate limiting at the end host in our current implementation lack precision. Real-world deployments implement very fine-grained rate limiters [75].

5.3 cosy Large-Scale Simulation

We now evaluate cosy using simulations on a datacenter-scale to complement the implementation results on our small-scale testbed. We use the standard 144-node leaf-spine topology [10,43,61]: it has 9 top-of-rack (ToR) switches, each connected to 16 end-hosts with access link bandwidth is 100Gbps.


(a) Max Sustainable Loads


(b) Slowdown Spreads


(c) Varying RPC Sizes

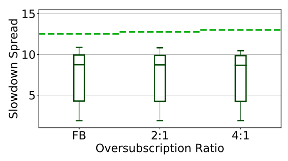

(d) Varying Oversubscription Ratio

Figure 5: Large scale simulation results: (a-c) sustainable loads and observed latency bounds showcase expectedly similar trends as seen on our testbed evaluations, (d) `cosy` sustains its performance even across oversubscribed topologies.

The propagation and switching delay are 200ns and 450ns, respectively. We use switch buffer capacity of 32MB (based on the commercially available switches [3–5]). The hRTT and bandwidth-delay product values for this topology are 5.5μs and 67KB, respectively. Figure 5 shows the simulation results. We observe that even on a datacenter-scale topology, the sustainable load and latency bound trends look almost identical to the ones obtained on our testbed.

6 Application Layer Benefits

We now present application layer benefits of communication synchrony using multiple applications.

6.1 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. We find that `cosy`'s predictable latency guarantees enables a fundamentally new point in the design space of CPU-efficient storage stacks. 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 still have suboptimal cpu-efficiency due to frequent context-switches. Exploiting `cosy` communication synchrony guarantees enables a new design point which achieves a new operating point — improved CPU efficiency without sacrificing much tail latency.

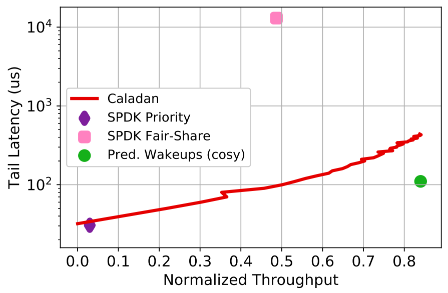


Figure 6: `cosy` **enables a new operating point in the design of CPU-efficient storage stacks.** Y-axis shows 99.9p latency of latency sensitive app, and X-axis shows throughput of throughput-bound app (Discussion in §6.1). Towards right and down is better.

Methodology and setup. We begin by analyzing the operating points achieved by today's storage stacks. We use SPDK [82], state-of-the art widely deployed storage stack, as representative of a polling-based system. We use Caladan [34], a recent kernel-bypass stack as representative of an interrupt-based system [5]. Our experimental setup consists of two servers (A, B) directly connected to a switch with 100Gbps links. We use two applications on server A (1) an I/O bound latency-sensitive app (LS) app which performs random reads to a remote storage device on server B (2) a CPU-intensive throughput-bound app (TB) which simply burns CPU cycles spinning in a while loop. We use a RAM disk as the remote storage device in order to emulate low-latency storage. For LS, we use an I/O size of 4KB, and an I/O depth of 8. We co-locate both LS and TB on a single CPU core in order to understand performance when applications share CPU resources. We give Caladan an additional CPU core for it's IOKernel. We measure 99.9th percentile tail latency of LS and the throughput achieved by TB (number of spins per second).

Polling-based designs. With SPDK, when LS is run in isolation, it achieves a very low tail latency of 29.8μs. When LS and TB are co-located (SPDK-fairshare in Figure 6) we observe a sigficant inflation in tail latency of LS, and a degradation in the throughput of TB (relative to when it is run in isolation). This is behavior is explained in prior work [46]. A different operating point can be achieved, by assigning LS higher CPU scheduling priority. This is shown with the SPDK (priority) data point in Fig. 6. LS achieves good tail latency, but at the cost of almost entirely starving TB.

Interrupt-based designs. When we run the experiment with Caladan in it's default configuration, it achieves good tail latency but poor throughput due to frequent context-switches — TB is repeatedly preempted when interrupts are delivered to LS. Caladan exhibits a tunable trade-off between latency and throughput in this scenario by enabling interrupts to be delayed and coalesced in order to minimize context-switch overheads. It exposes a parameter, `THRESH_QD` for this. We re-run the experiment with varying `THRESH_QD` Fig. 6. As `THRESH_QD` is graudually increased the throughput increases at the cost of higher tail latency. We begin to see diminishing

[5]While Caladan's IOKernel is polling-based, the applications themselves are interrupt-driven

returns for larger THRESH_QD values.

Predictable-wakeups. The asynchronous nature of existing networks fundamenatally constrains the design space of storage stacks. cosy enables a new design point via it's property of communication synchrony — *Predictable Wakeups*. A storage stack using cosy as transport can predict precisely the time bound within which response packets will arrive (based on cosy's latency bound) This enables it to put the application to sleep and wake it up at the right point in time so as to minimize tail latency and maximize CPU-efficiency. We integrated cosy as a new transport in SPDK, and modified the storage stack logic to support predictable wakeups.

Evaluation. We evaluate our prototype of predictable wakeups using the same experimental setup as before. We configure cosy with $K = 8$. The resulting operating point is shown in Figure 6. Compared to polling-based systems, cosy + predictable wakeups is able to achieve significantly better throughput with only minimal degradation in tail latency. TB achieves higher throughput because LS goes to sleep right after issuing requests, enabling TB to use CPU cycles until LS is woken up again. LS tail latency increase slightly because of (1) rsvp/complete exchange in cosy[6] (2) waiting for responses to all X requests to arrive before waking up.

Compared to interrupt-based systems, cosy + preditable wakeups is able to achieve higher throughput without compromising significantly on tail latency. Let us focus on two points on the Caladan frontier in Fig. 6: (1) The point at which Caladan achieves the same tail latency as cosy + predictable wakeups. Here Caladan's throughput is 37% lower. (2) The point where Caladan achieves the same throughput as cosy + predictable wakeups. Here it's tail latency is nearly 4× higher. The reason for this is the timeout-based nature of interrupt coalescing (via Caladan's THRESH_QD parameter). Since packet arrival times are not predictable in asynchronous networks, using a fixed timeout value leads to either (1) poor throughput if the timeout happens to be too low or (2) high tail latency if the timeout happens to be too high. Hence, we see that compared to both polling-based and interrupt-based systems on asynchronous networks, cosy enables achieveing a new and better operating point that was previously not possible.

6.2 Efficient Packet Processing Pipeline

Multiple recent works [21, 37] have suggested that running multiple applications/connections on a single CPU core can result in significant degradation in application throughput or CPU efficiency. This is due to larger number or cache misses caused by multiple contending apps. A recently introduced design – Reframer [37], tries to reduce these misses by deliberately buffering packets, waiting for a batch worth of packets to arrive per application, before letting them processed by the CPU. Hence, a key requirement for such a design to provide benefits is for requisite number of packets to arrive in as many batches as possible. However as discussed in this paper before, for existing asynchronous networks arriving packets can be arbitrarily delayed. In the presence of large packet inter-arrival delays, one could potentially use a large buffering timeouts to allow for desired batch sizes. However, that would result in increased latencies for arriving packets as they would require being buffered for larger amount of time.

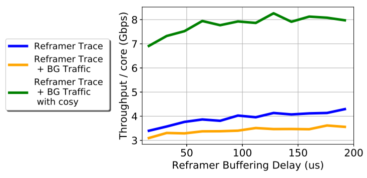
Such an application presents a perfect case for the need of communication synchrony guarantees provided by cosy. cosy provides very predictable inter-packet arrival times for its traffic, and users can easily set a desired buffering timeout to reap the maximum benefits of a scheme like Reframer.

We evaluate the benefit of using cosy with Reframer using the suggested two server in-chain setup as employed in the Reframer paper [37]. We use two servers directly connected to each other with a 100Gbps link. The network functions and Reframer implementations are based on FastClick – a DPDK-based framework for network I/O [17]. Reframer implementation takes as input packet capture dump files, and replays the trace using the time of arrival and 5-tuple information for each packet. We use the traces provided in the Reframer repository as the baseline [7]. This trace was captured with TCP as underlying transport, and hence packets can experience large variation in inter-packet arrival times like in any asynchronous network. In order to fully capture the network asynchrony that can be experienced on a large scale topology in the presence of background traffic, we also add randomized delays to inter-packet arrival delays, while keeping the average throughput of each flow in the traces the same. We run the same traces on cosy and capture the corresponding traffic, to replay on Reframer implementation in order to evaluate Reframer performance in presence of cosy. Note that since the background traffic is sent using the asynchronous interface of cosy, it will have no effect on the synchronous traffic performance due to priority-based isolation as discussed previously in §4.

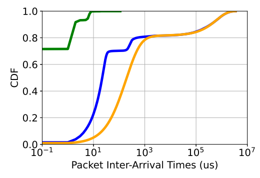
Figure 7a shows the increase in average throughput per core using Reframer by increasing the buffering timeout values from 16μs to 192μs. We see that Reframer improves the throughput per core from 3.4 Gbps to 4.2 Gbps for the baseline trace. With the introduction of background traffic induced delays, the throughput per core is reduced for each corresponding timeout value. Using cosy Reframer is able to achieve 6.9 Gbps throughput per core even with the 16μs of bufferring.

Figures 7b-7d explain how cosy allows Reframer to achieve these benefits. Figure 7b shows inter-packet delay profiles of the evaluated traces. We plot the distribution of packet inter-arrival times across all flows in the traces. We see that cosy indeed provides very consistent and low inter-packet arrival times, with almost all inter-arrival times $\leq 10\mu$s. cosy provides average batch size of 4.4 for 16μs buffering timeout, and is able to achieve close to best possible average

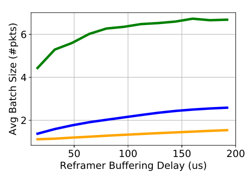
[6]Note, that we treat each I/O request as a separate cosy message



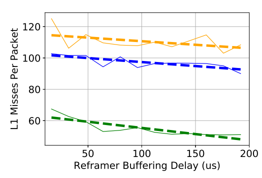
(a) Avg Throughput Per Core



(b) Delay profile of the trace

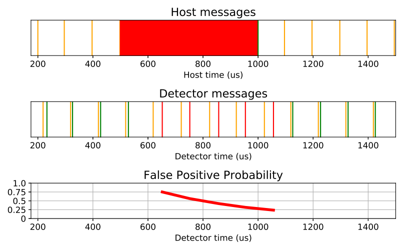


(c) Reframer Avg Batch Size

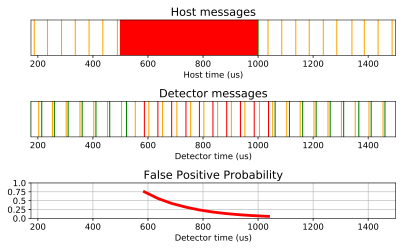


(d) L1 Misses Per Second

Figure 7: **Using cosy to improve the effectiveness of Reframer:** (Discussion in § 6.2)



(a) Detector using T_P = 100μs



(b) Detector using T_P = 50μs

Figure 8: **Using cosy to design efficient endhost failure detector (See §6.3).** Top plots show `requests` arriving at the endhosts from the detector and the times when the endhost is crashed; middle plots show `requests` being sent from the detector and the corresponding `rsvps` arriving from the endhost when not crashed; bottom plots show the false positive probability reducing with more consequtive unresponded `requests`.

batch size for the evaluated traces within timeout of ∼128μs. Without cosy however, Reframer is only able to achieve average batch size of 2.5 (and 1.5 in the presence of background traffic) even with buffering timeout of 192μs. As discussed previously, larger batching provides better CPU efficiency due to improved cache miss rates. Figure 7d shows reduction in L1 cache miss rates per packet with increasing buffering timeouts, and this reduction is proportional to the increase throughput per core.

6.3 Efficient Failure Detectors

Detecting host failures effectively is a fundamental problem in distributed systems. Designing a failure detector which is reliable (i.e. provides small false positive probability) and fast (provides failure notification within a small delay) is a hard problem, especially in the presence of today's unpredictable networks [55]. However, one can achieve both these goals efficiently utilizing cosy synchrony guarantees.

To demonstrate this, we implemented our own endhost failure detector application on top of cosy: a detector endhost periodically sends a `request` to the desired endhost every fixed interval of probing time T_P. If the detector hears back a `rsvp` from the endhost, it assumes the endhost is alive, otherwise if there is no response for δ_{cosy} worth of time, the detector concludes the endhost is crashed. In the presence of no other network failure between the path of the detector and endhost, cosy would ensure that the detector receives `rsvp` within δ_{cosy} time if the endhost remains non-faulty. However, there is a chance of the detector raising false positives in the presence of link failures since `request` could be dropped on the failed links and never reach the endhost. Due to the large path diversity available in typical datacenter topologies the probability of false positives is already low since the `request` and `rsvp` would usually have large number of non-faulty network paths available.

Figure 8 shows the evaluation results for this failure detector implemented on our testbed setup shown in Figure 4, with one endhost used as a detector and one which can crash. Figure 8a shows the detector sending a `request` to the endhost every 100us (shown in orange). The endhost is made to crash at 500us and recovered back at 1000us (shown in red). When alive, the endhost responds back to the detector using `rsvp` (shown in green). As guaranteed by cosy, all `rsvp` eventually arriving at the detector arrive within δ_{cosy} period of sending a `request`. Upon crash, the detector can detect it within a maximum duration of $T_P + \delta^E$, which is only arround $\sim 135\mu$s for the setup. Figure 8b shows the corresponding result with $T_P = 50\mu$s. Expectedly the maximum failure response time reduces to 85μs. Hence, increasing the probing frequency allows quicker detection of failures with by trading off more network bandwidth employed by the detector.

7 Conclusion

Applications today use best-effort delivery semantics. Such weak semantics reduces the set of assumptions that system designers can rely upon from the underlying network, thus introducing complexity and inefficiency in end-host systems. Motivated by recent hardware and application trends, this paper explores design and performance tradeoffs for networks offering partial communication synchrony semantics.

References

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

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

[3] Barefoot Networks. https://www.barefootnetworks.com/.

[4] Broadcom bcm56850. https://www.broadcom.com/collateral/pb/56850-PB03-R.pdf.

[5] Cisco Switches. https://www.cisco.com/c/en/us/products/switches/.

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

[7] Reframer implementation. https://github.com/hamidgh09/Reframer.

[8] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. In *ACM SIGCOMM*, 2008.

[9] 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.

[10] 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.

[11] M. T. Arashloo, A. Lavrov, M. Ghobadi, J. Rexford, D. Walker, and D. Wentzlaff. Enabling programmable transport protocols in high-speed NICs. In *USENIX NSDI*, 2020.

[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] W. Bai, L. Chen, K. Chen, D. Han, C. Tian, and W. Sun. Pias: Practical information-agnostic flow scheduling for data center networks. In *Proceedings of the 13th ACM workshop on hot topics in networks*, pages 1–7, 2014.

[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 Transactions on networking*, 2000.

[17] T. Barbette, C. Soldani, and L. Mathy. Fast userspace packet processing. In *2015 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, pages 5–16. IEEE, 2015.

[18] C. Basso, J. Calvignac, D. Orsatti, and F. Verplanken. Hop-by-hop flow control in an ATM network, 1998. US Patent 5,787,071.

[19] A. Belay, G. Prekas, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion. IX: A protected dataplane operating system for high throughput and low latency. In *USENIX OSDI*, 2014.

[20] Q. Cai, S. Chaudhary, M. Vuppalapati, J. Hwang, and R. Agarwal. Understanding host network stack overheads. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, pages 65–77, 2021.

[21] Q. Cai, S. Chaudhary, M. Vuppalapati, J. Hwang, and R. Agarwal. Understanding host network stack overheads. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, pages 65–77, 2021.

[22] I. Cho, K. Jang, and D. Han. Credit-scheduled delay-bounded congestion control for datacenters. In *ACM SIGCOMM*, 2017.

[23] R. Cole, K. Ost, and S. Schirra. Edge-coloring bipartite multigraphs in o (e log d) time. *Combinatorica*, 21(1):5–12, 2001.

[24] F. Cristian and C. Fetzer. The timed asynchronous distributed system model. In *IEEE Transactions on Parallel and Distributed systems*, 1999.

[25] W. J. Dally, P. P. Carvey, L. R. Dennison, and P. A. King. Router with virtual channel allocation, May 13 2003. US Patent 6,563,831.

[26] W. J. Dally and C. L. Seitz. Torus routing chip, June 12 1990. US Patent 4,933,933.

[27] D. Dolev, C. Dwork, and L. Stockmeyer. On the minimal synchronism needed for distributed consensus. In *JACM*, 1987.

[28] P. Dutta, R. Guerraoui, and L. Lamport. How fast can eventual synchrony lead to consensus? In *DSN*, 2005.

[29] T. S. El-Bawab and J.-D. Shin. Optical packet switching in core networks: between vision and reality. In *IEEE Communications Magazine*, 2002.

[30] 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.

[31] A. Farshin, T. Barbette, A. Roozbeh, G. Q. Maguire Jr, and D. Kostić. Packetmill: toward per-core 100-gbps networking. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1–17, 2021.

[32] A. Farshin, A. Roozbeh, G. Q. Maguire Jr, and D. Kostić. Reexamining direct cache access to optimize i/o intensive applications for multi-hundred-gigabit networks. In *2020 {USENIX} Annual Technical Conference ({USENIX}{ATC} 20)*, pages 673–689, 2020.

[33] 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.

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

[35] B. Gamlath, M. Kapralov, A. Maggiori, O. Svensson, and D. Wajc. Online matching with general arrivals. In *2019 IEEE 60th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 26–37. IEEE, 2019.

[36] 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.

[37] H. Ghasemirahni. *Packet Order Matters!: Improving Application Performance by Deliberately Delaying Packets*. PhD thesis, KTH Royal Institute of Technology, 2021.

[38] 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 *Networked Systems Design and Implementation (NSDI)*, 2022.

[39] M. Ghobadi, R. Mahajan, A. Phanishayee, N. Devanur, J. Kulkarni, G. Ranade, P.-A. Blanche, H. Rastegarfar, M. Glick, and D. Kilper. ProjecToR: Agile reconfigurable data center interconnect. In *ACM SIGCOMM*, 2016.

[40] A. Greenberg, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. Maltz, P. Patel, and S. Sengupta. VL2: A scalable and flexible data center network. In *ACM SIGCOMM*, 2009.

[41] 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.

[42] 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.

[43] 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.

[44] 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 *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 422–434, 2020.

[45] J. Hwang, Q. Cai, A. Tang, and R. Agarwal. TCP$\approx$RDMA: Cpu-efficient remote storage access with i10. In *NSDI*, 2020.

[46] J. Hwang, M. Vuppalapati, S. Peter, and R. Agarwal. Rearchitecting linux storage stack for μs latency and high throughput, 2021.

[47] V. Jacobson. Congestion avoidance and control. In *SIGCOMM*, 1988.

[48] E. Jeong, S. Woo, M. A. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park. mTCP: a highly scalable user-level TCP stack for multicore systems. In *USENIX NSDI*, 2014.

[49] A. Kalia, M. Kaminsky, and D. Andersen. Datacenter rpcs can be general and fast. In *USENIX NSDI*, 2019.

[50] R. Kapoor, G. Porter, M. Tewari, G. M. Voelker, and A. Vahdat. Chronos: Predictable low latency for data center applications. In *Proceedings of the Third ACM Symposium on Cloud Computing*, pages 1–14, 2012.

[51] A. Kaufmann, T. Stamler, S. Peter, N. K. Sharma, A. Krishnamurthy, and T. Anderson. TAS: TCP acceleration as an OS service. In *EuroSys*, 2019.

[52] S. Keshav and S. Kesahv. *An engineering approach to computer networking: ATM networks, the Internet, and the telephone network*, volume 116. Addison-Wesley Reading, 1997.

[53] 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 *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications,*

technologies, architectures, and protocols for computer communication, pages 514–528, 2020.

[54] 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.

[55] 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 *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 279–294, 2011.

[56] J. Li, E. Michael, N. K. Sharma, A. Szekeres, and D. R. Ports. Just say NO to paxos overhead: Replacing consensus with network ordering. In *USENIX OSDI*, 2016.

[57] 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 *ACM SIGCOMM*. 2019.

[58] 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 *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 399–413, 2019.

[59] 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.

[60] 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.

[61] B. Montazeri, Y. Li, M. Alizadeh, and J. Ousterhout. Homa: A receiver-driven low-latency transport protocol using network priorities. In *ACM SIGCOMM*, 2018.

[62] C. S. R. Murthy and M. Gurusamy. *WDM optical networks: concepts, design, and algorithms*. Prentice Hall, 2002.

[63] R. Neugebauer, G. Antichi, J. F. Zazo, Y. Audzevich, S. López-Buedo, and A. W. Moore. Understanding pcie performance for end host networking. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 327–341, 2018.

[64] G. T. Nguyen, R. Agarwal, J. Liu, M. Caesar, P. B. Godfrey, and S. Shenker. Slick packets. *ACM SIGMETRICS Performance Evaluation Review*, 39(1):205–216, 2011.

[65] 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.

[66] C. Özveren, R. Simcoe, and G. Varghese. Reliable and efficient hop-by-hop flow control. In *ACM SIGCOMM*, 1994.

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

[68] S. Peter, J. Li, I. Zhang, D. R. Ports, D. Woos, A. Krishnamurthy, T. Anderson, and T. Roscoe. Arrakis: The operating system is the control plane. *ACM Transactions on Computer Systems (TOCS)*, 33(4):1–30, 2015.

[69] S. Plotkin. Competitive routing of virtual circuits in atm networks. *IEEE Journal on Selected Areas in Communications*, 13(6):1128–1136, 1995.

[70] 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. *ACM SIGCOMM Computer Communication Review*, 43(4):447–458, 2013.

[71] D. R. Ports, J. Li, V. Liu, N. K. Sharma, and A. Krishnamurthy. Designing distributed systems using approximate synchrony in data center networks. In *Usenix NSDI*, 2015.

[72] 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 *ISCA*, 2014.

[73] J. Rexford. Tailoring router architectures to performance requirements in cut-through networks. 1999.

[74] S. M. Rumble, D. Ongaro, R. Stutsman, M. Rosenblum, and J. K. Ousterhout. Its Time for Low Latency. In *ACM HotOS*, 2011.

[75] A. Saeed, N. Dukkipati, V. Valancius, V. The Lam, C. Contavalli, and A. Vahdat. Carousel: Scalable traffic shaping at end hosts. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 404–417, 2017.

[76] 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.

[77] 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 *SIGCOMM Comput. Commun. Rev.*, 2015.

[78] P. Stuedi, A. Trivedi, B. Metzler, and J. Pfefferle. Darpc: Data center rpc. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 1–13, 2014.

[79] A. Tootoonchian, A. Panda, C. Lan, M. Walls, K. Argyraki, S. Ratnasamy, and S. Shenker. Resq: Enabling slos in network function virtualization. In *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*, pages 283–297, 2018.

[80] B. C. Vattikonda, G. Porter, A. Vahdat, and A. C. Snoeren. Practical tdma for datacenter ethernet. In *EuroSys*, 2012.

[81] T. Yang, R. Gifford, A. Haeberlen, and L. T. X. Phan. The synchronous data center. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, pages 142–148, 2019.

[82] Z. Yang, J. R. Harris, B. Walker, D. Verkamp, C. Liu, C. Chang, G. Cao, J. Stern, V. Verma, and L. E. Paul. Spdk: A development kit to build high performance storage applications. In *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 154–161. IEEE, 2017.

[83] L. Zhang, S. Berson, S. Herzog, S. Jamin, and R. Braden. RFC2205: Resource ReSerVation Protocol (RSVP) – version 1 functional specification, 1997.

[84] L. Zhang, S. Deering, D. Estrin, S. Shenker, and D. Zappala. Rsvp: A new resource reservation protocol. *IEEE network*, 7(5):8–18, 1993.

[85] 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.

A Proofs for theorems

Proof for Theorem 3.1. Assuming a 2-tier leaf-spine and a full-bisection topology, we can represent the topology as shown in Figure 9 (in an unfolded view, with senders on the left side and receivers on the right) . To ensure communication synchrony guarantee, the network must provide two necessary conditions – (1) $\exists$ an $F > 0$ such that each message admitted by the network must transmit all its packets at an average transmission rate $\geq B/F$, where B is the access link bandwidth for each endhost in the network; and (2) all the switch ports must receive data at average rates smaller than the bandwidth capacity for the respective switch port. If (1) is not satisfied, the transmission delay for the message can be arbitrarily high leading to unbounded latency. If (2) is not satisfied, this could lead to unbounded queueing at the switch buffer, leading to packet drop[7].

To provide optimal latency, the network must transfer the message using rate B (i.e., $F = 1$). We now see how one can realize condition (2) for such a network. We represent the topology using a bipartite graph $\mathcal{G}$ formed using senders and receivers as the left and right set of vertices respectively (bottom-left, Figure 9). We represent the input traffic as edges on this graph, with edges formed between the message sender and receiver vertices. Consider a graph $\mathcal{M}$ formed by performing bipartite matching on $\mathcal{G}$ (bottom-center, Figure 9). We then convert $\mathcal{M}$ to a bipartite multigraph $\mathcal{L}$ by using leaf-switches (or the ToR switches) instead of endhosts as the vertices (top-center, Figure 9). Next, we perform edge-coloring on $\mathcal{L}$ to obtain an edge-colored multigraph $\mathcal{Z}$ (top-right, Figure 9). Note that for bipartite multigraphs with degree d, it is known that we can always optimally edge-color the graph using d colors [23]. In this case, each unique color can be mapped to a unique spine switch (or the core switch). It can be easily seen that if the network only admits set of messages admitted by the matching in $\mathcal{M}$ and forwards the messages at full rate via the spine switches corresponding to the color assigned to each message edge in $\mathcal{Z}$, we obtain a synchronous network – the network satisfies the condition (2) above, since each message takes an edge-disjoint path in the network, and each switch port in a full-bisection bandwidth topology has capacity equal to the access link bandwidth.

The sources of latency for any message are (a) waiting delay for admission (b) transmission delay at the sender (c) speed-of-light propagation delay for all links and crossbar fabric switching delay for all switches across the network and (d) queueing delay at each switch. The required edge-coloring algorithm could be performed either in a distributed or a centralized manner; either scenario would require at least one round of communication between a pair of nodes in the network. Hence, the arriving message waits for minimum one hRTT (recall that one hRTT is equal to the round trip delay for packet across the network and includes link propagation delays and crossbar switching deelays) before it can be admitted. Upon admission, the sender transmits the packets using rate B, therefore the transmission time for a message with size s equals s/B. Since for the current scenario with $F = 1$, each message is assigned an edge-disjoing path in the network, there is zero queueing at each switch, hence there is no queueing delay observed by any packet in the network. Assuming the scenario of optimal latency, i.e., message arrived at time $t = 0$ at the sender, and got admitted at $t = 1$hRTT and started

[7] One can also easily construct scenarios where using hop-by-hop control techniques like PFC [1] in order to avoid packet drops in presence of (2) could lead to violation of (1) in the worst-case

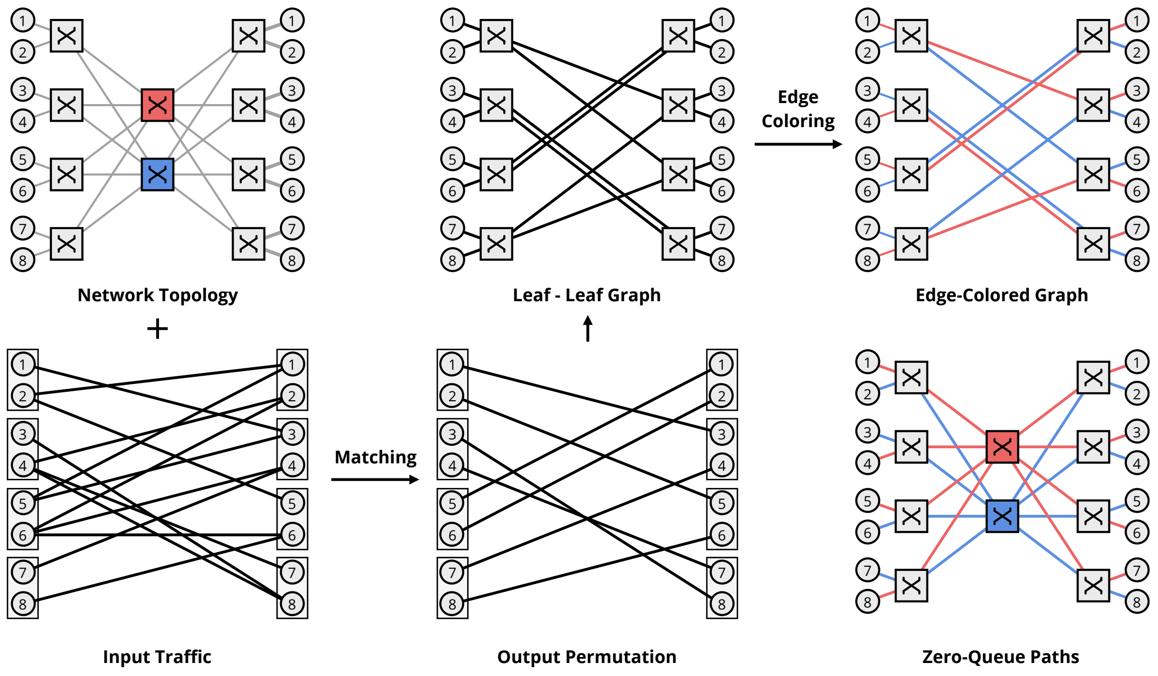


Figure 9: Constructing a network providing communication synchrony with the help of graph matchings and edge-coloring

sending its data, then the receiver receives all the required data packets from the sender by time $t = s/B + \frac{3}{2}$hRTT. The extra 0.5hRTT arises due to the propagation and switching delay for the last packet traversing from the sender to the receiver. It takes another 0.5hRTT to notify the sender about message arrivals or possible hardware failure (in case the message does not arrive within intended time at the receiver). Hence by the time $\delta^\star = 2$hRTT + s/B, the sender completion queue receives the information about the message being completed or rejected. Hence latency experienced by any network providing communication synchrony $\geq \delta^\star$.

For the current scenario of $F = 1$, an admitted edge is added to the matching irrevocably. The utilization of the network is the total data transmitted by the network per unit time. The utilization of the network is directly proportional to the number of edges admitted; the maximum utilization occurs when the network admits the edges corresponding to the offline maximum matching. The network however performs a *maximal* matching under the model of edge-arrivals without preemption, and the problem is known to be strictly $\frac{1}{2}-$competitive [35]; hence there exists a input traffic pattern such that the utilization $= \frac{1}{2(1+\alpha)}$. The factor of $\frac{1}{(1+\alpha)}$ arises due to fact that no data is transmitted by the sender during the first hRTT when the sender waits to be informed whether the message is admitted by the network. For generic scenario of $F > 1$, the utilization is proportional to a fractional matching, where the weight associated with each admitted edge is proportional is equal to $R/B \geq 1/F$, where R is the rate at which network transmits the data for the message corresponding to the admitted edge (recall that $B \geq R \geq B/F$). [35] shows any algorithm for performing maximal fractional matching under the edge-arrival model is also $\frac{1}{2}-$competitive. Hence the utilization bounds remains the same. Therefore, for any network providing communication synchrony there exists a traffic matrix such that the utilization $\Theta \leq \Theta^\star = \frac{1}{2(1+\alpha)}$. Θ is also the maximum sustainable load by the network, since by definition maximum sustainable load will be equal to the network utilization for a scenario where optimal utilization $= 1$, and the bound on Θ is independent of optimal utilization.

Proof for Theorem 3.2 and 3.4. We now provide bound for utilization in expectation across randomized request forwarding decisions. If the input workload is matching to begin with, an edge is not admitted by `cosy` iff there is an edge-coloring conflict – i.e., `cosy` chooses a spine for forwarding the arrived message (corresponding to the edge) which already has K virtual links reserved for the sender-side leaf – spine link. To find the expected the number of edges admitted using `cosy`, we employ the commonly used balls-and-bins argument in our context.

Consider any sender-side leaf switch. Each leaf – spine link is considered a bin. Each arriving message is considered a ball. Whenever `cosy` forwards the message request via a randomly chosen spine, we get a new ball assigned to the bin corresponding to the spine. Hence, whenever a new message arrives, we correspondingly get a new ball assigned to a bin. For any bin, we consider a maximum of K balls to be "admitted" (since at most K virtual links are allowed per physical link). However, since `cosy` assigns balls to a bin uniformly

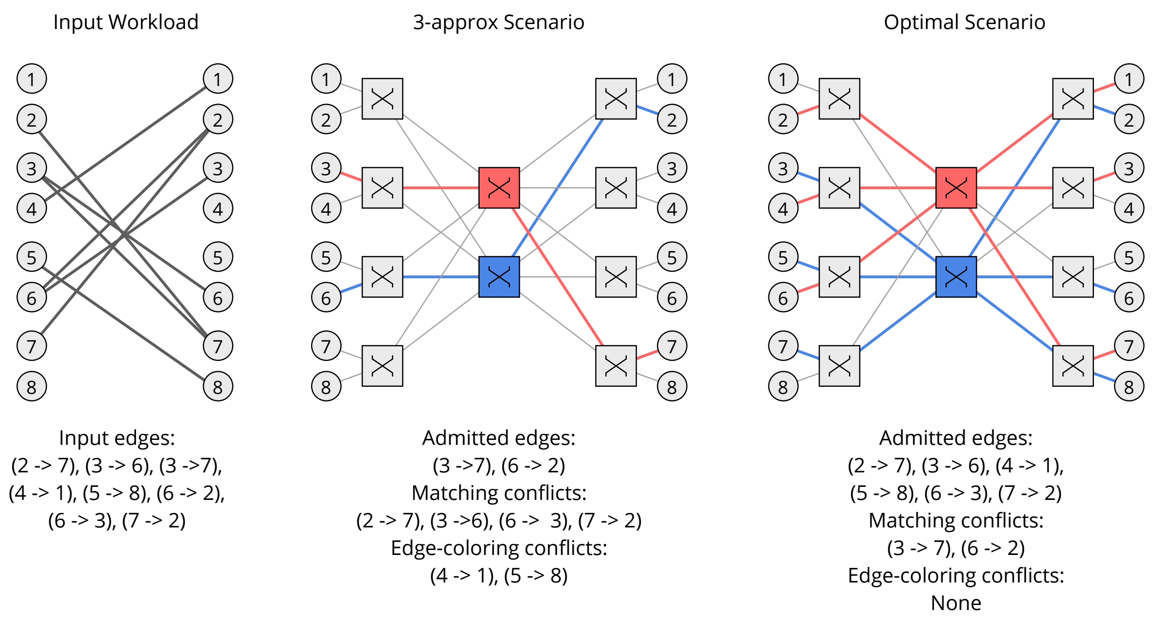


Figure 10: An example of RSVP admitting edges $1/3\times$ the optimal. The example uses the unfolded view a two-tier leaf-spine topology with 8 nodes (as used in the proof for Theorem 3.1). If RSVP admits edges $(3 \rightarrow 7)$ and $(6 \rightarrow 2)$ and chooses the colors as for these edges as shown, then no other edge can be admitted, either due to a matching constraint or due to an edge-coloring constraint (discussed in proof for Theorem 3.1). However, optimally it is possible to admit 6 edges as shown on right hand side.

randomly, a bin can be assigned any number of balls (with varying probabilities). We now find the probability p_t that t balls are assigned to any bin and then use this to find the expected number of balls admitted by any bin.

Assume that there are a total of N input balls and C bins, with each ball arriving one after the other without loss of generality. The probability that any bin will have exactly t balls is

$$p_{t,N} = \binom{N}{t}\left(\frac{1}{C}\right)^t\left(1-\frac{1}{C}\right)^{(N-t)} \tag{2}$$

The expected number of balls admitted by any bin is

$$b_N = \sum_{i=0}^{N} \min(i,K)\, p_{i,N} \tag{3}$$

The fraction of expected number of edges admitted f is given by bC/N. It can be easily seen that f strictly decreases with increasing value of N, since after each edge admission the probability that the newly arrived edge would be rejected strictly increases. Since for any leaf the maximum value of N is bounded by KC (since we assume input as matching, and the topology is full-bisection), f is also bounded by $f(K,C)$

$$f \geq f(K,C) = b_{KC} \times C/KC \tag{4}$$

We now use the above result to prove that cosy provides $\frac{f(K,C)}{2(1+\alpha)}$-competitive performance in expectation for arbitrary input traffic patterns similar to the previous proofs. Assume the offline optimal matching size is $M^\star$, and the number of edges finally admitted by cosy is $< \frac{M^\star f(K,C)}{2}$. Result above shows that for any input matching, cosy admits $\geq M^\star f(K,C)$ edges in exptection. Hence, if there wasn't any coloring constraint for admitting the edges by cosy, the input matching size would've been $< \frac{M^\star}{2}$. Note that without the coloring constraint, the scheme would simply construct a maximal matching, which is always $\geq \frac{M^\star}{2}$. Hence, a contradiction and our assumption that the expected number of edges finally admitted by cosy is $< \frac{M^\star f(K,C)}{2}$ is false. Using result from Theorem 3.1, we get the required result.

Proof for Lemma 3.3. Consider any arbitrary switch in the topology. cosy ensures that data packets corresponding to at most K messages, each possibly from a different input port (if the number of input ports in the switches $\geq K$), arriving at the same output port concurrently. Since cosy senders ensure that packets from each slot are sent at the rate $1/(K)$, there will be no persistant queueing.

However, there can be queue-build up happening because of temporary contention arising at the output port – it is possible that packets from multiple input ports (belonging to different slots) intending to be forwarded via same output port, arrive at their respective input ports at the same time. Hence, some packets would have to be enqueued before they can be forwarded. It can be shown that at the first network hop the maximum queue-size can be $(K-1)$ packets, happeneing when the all K packets arrive at their input ports at the same time. Due to this queueing at the first network hop, the pacing

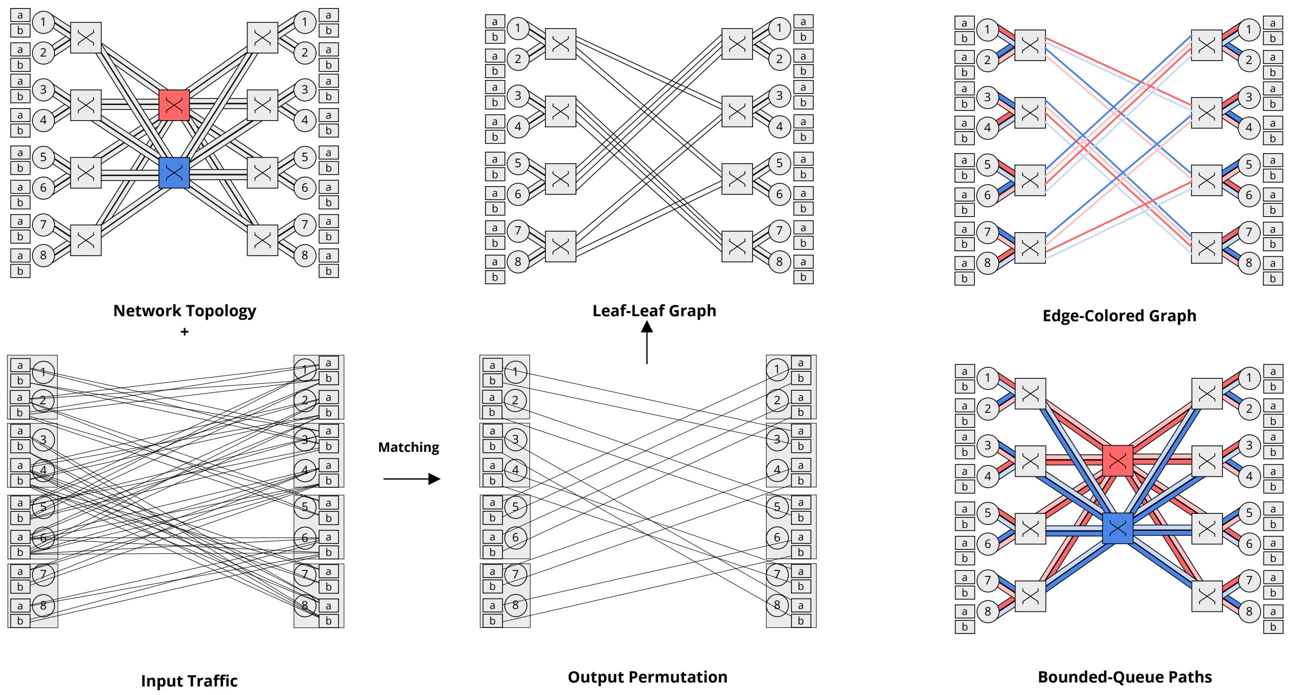


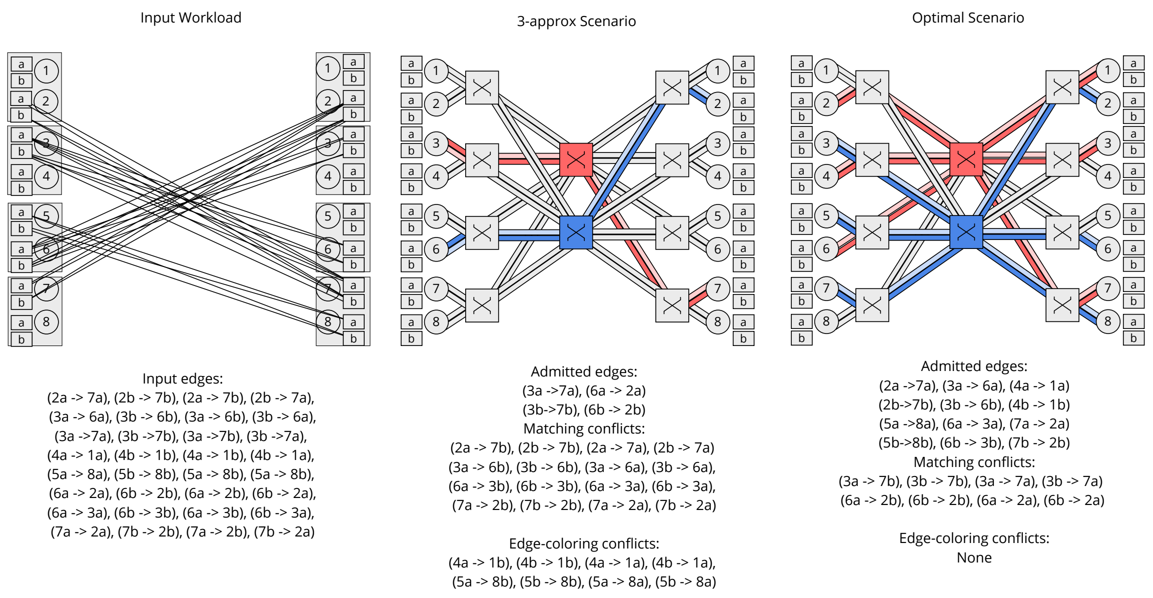
Figure 11: Extending the graph construction to show matching and edge-coloring for the case of $K = 2$. For constructing the matching, each endhost slot – ffor eg. here Na and Nb in this figure are used as the vertices for any endhost N, and we construct edges $\{$Na – Ma, Nb – Mb, Na – Mb, Nb – Ma$\}$ for each message arriving between (N,M). After performing matching upon this graph, we get a leaf-leaf multigraph with degree $K\times$ the degree of the corresponding multigraph obtained in Figure 9, hence we use $K\times$ the colors for coloring this multigraph. The messages now assigned first K colors are routed via first spine, second K colors via second spine, and so on. For this figure, we use light red and red colors routed via red spine, and light blue and blue colors routed via blue spine. Note that we use the same input workload as in Figure 9 for this example.

of packets arriving at the next-hop gets altered – although the packets belonging to a given virtual path still arrive at an average rate of $(1/K)\times$ the link bandwidth, the instantaneous rate can possibly increase for a brief instant of time. It can be seen that with the addition every hop, in the worst-case we can see additional queueing of $(K-1)$ packets per port – leading to the maximum possible queueing of $H\times(K-1)$ per port per switch , where H is the number of hops in the topology. Figure 13 shows an example workload which can create a queueing of $3{*}(K-1)$ at a switch port at third hop.

Proof for Theorem 3.5. For networks having large enough switch capacities, there would be no packet drops due to buffer overflow and therefore, bounded delays for each traversed message.

The packets in cosy experience larger delays than the basic design due to two reasons – lower transmission rates due to slot allocation and larger queueing delays. If a message has been only allocated a single slot, the transmission time for message of size m would be mK/B. Regarding queueing delay, as discussed previously, in cosy i^{th} hop switch would experience maximum per-port queueing of $i\times(K-1)$ packets, and the queues will be drained at bandwidth B, resulting in a total maximum possible delay of $\frac{\#H\times(\#H+1)\times(K-1)\times p}{(2\times B)}$, which is equal to $\hat{\delta}$. Therefore the total delay between the source and receiver buffers seen by any message in cosy is $\leq mK/B +$ hRTT$/2+\hat{\delta}$. Additionally, cosy allows request to wait for δ^E at the receivers, hence it can take hRTT + δ^E delay for a message to get admitted into the network, and 1/2 hRTT for the receiver to notify the sender about message arrival or possible failure.

Handling non-homegeneous topologies. cosy also provides communication synchrony in the presence of a non-homegeneous topology, with link bandwidth not necesarily the same across the topology. Consider any arbitrary switch inside the network. Assume the output port bandwidth is larger than input port – let's say $F\times$ the input port bandwidths. The design ensures that data packets from *at most* $K\times F$ different input ports be forwarded to the same output port, with the packets arriving from each input port at rate $1/(K\times F)$ times the output port bandwidth. This ensures that there would be no persistent queue-build up since the net packet arrival rate is same as the net packet departure rate. However, as shown in proof for Lemma 3.3, the maximum queue build-up at the i^{th} hop can be $i\times(KF-1)$ packets, and the delay due to



Input edges:
(2a -> 7a), (2b -> 7b), (2a -> 7b), (2b -> 7a),
(3a -> 6a), (3b -> 6b), (3a -> 6b), (3b -> 6a),
(3a ->7a), (3b ->7b), (3a ->7b), (3b ->7a),
(4a -> 1a), (4b -> 1b), (4a -> 1b), (4b -> 1a),
(5a -> 8a), (5b -> 8b), (5a -> 8b), (5a -> 8b),
(6a -> 2a), (6b -> 2b), (6a -> 2b), (6b -> 2a),
(6a -> 3a), (6b -> 3b), (6a -> 3b), (6b -> 3a),
(7a -> 2a), (7b -> 2b), (7a -> 2b), (7b -> 2a)

Admitted edges:
(3a ->7a), (6a -> 2a)
(3b->7b), (6b -> 2b)
Matching conflicts:
(2a -> 7b), (2b -> 7b), (2a -> 7a), (2b -> 7a)
(3a -> 6b), (3b -> 6b), (3a -> 6a), (3b -> 6a),
(6a -> 3b), (6b -> 3b), (6a -> 3a), (6b -> 3a),
(7a -> 2b), (7b -> 2b), (7a -> 2a), (7b -> 2a)

Edge-coloring conflicts:
(4a -> 1b), (4b -> 1b), (4a -> 1a), (4b -> 1a),
(5a -> 8b), (5b -> 8b), (5a -> 8a), (5b -> 8a)

Admitted edges:
(2a ->7a), (3a -> 6a), (4a -> 1a)
(2b->7b), (3b -> 6b), (4b -> 1b)
(5a ->8a), (6a -> 3a), (7a -> 2a)
(5b->8b), (6b -> 3b), (7b -> 2b)
Matching conflicts:
(3a -> 7b), (3b -> 7b), (3a -> 7a), (3b -> 7a)
(6a -> 2b), (6b -> 2b), (6a -> 2a), (6b -> 2a)

Edge-coloring conflicts:
None

Figure 12: For K > 1, we can also achieve $1/3-$competitive performance as shown previously for the case of $K = 1$ (or RSVP). The current figure shows the extension of the example in Figure 10 with $K = 1$ to the case of $K = 2$. We can easily extend the same example for larger values of K as well in a similar fashion.

queueing at this hop is $i \times (KF-1) \times p/(FB)$. Hence, the total delay by queueing is still bounded by $(\#H) \times (\#H+1) \times (KF-1) \times p/(2FB)$.

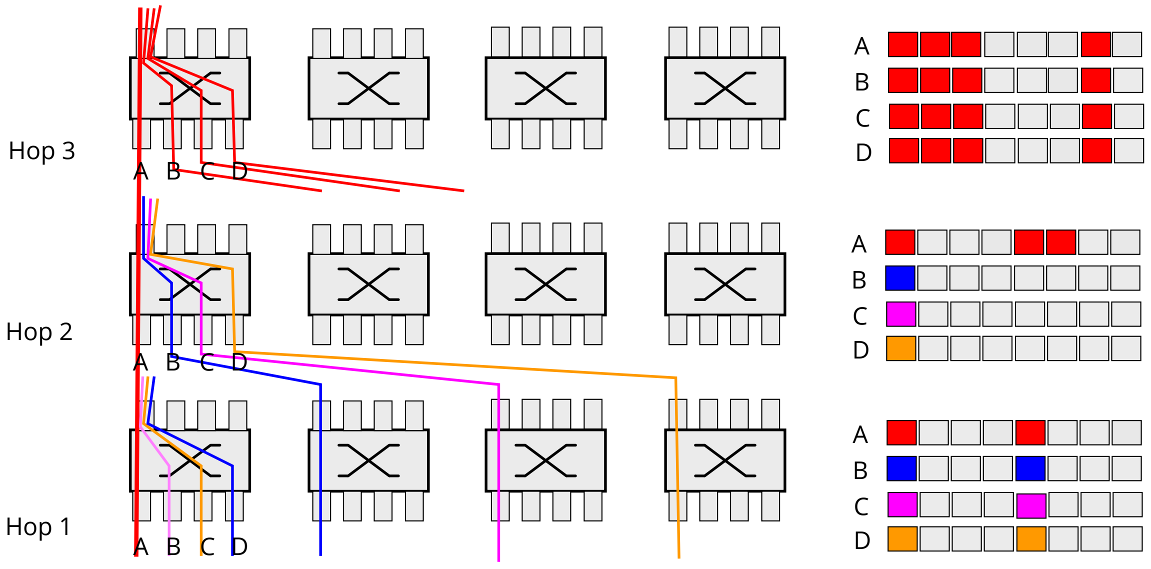


Figure 13: An example showing that that maximum queueing at hop i can be $i \times (K-1)$ packets. In this example we see a subset of switches in a topology and $K = 4$, and different colors depicts the virtual paths taken by different message streams, each sending data packets at average rate of $B/4$. The right side of figure shows the packets for different message streams arriving at respective input ports at the switches. As shown, hop 1 receives perfectly paced packets, but leads to maximum queueing of 3 packets when all red, blue, pink and orange packets arrive at the input ports at the same time. Due to queueing at hop 1, the packing for packets can get disturbed, as shown for red packets at hop 2 input port, it is possible to have 2 red packets next to each other (if at hop 1, the red packet was the last to be dequeued amongst the 3 enqueued packets). Now at the other input input ports of hop 1, suppose single packets corresponding to other messages arrive exactly at the same time, the red packet can again be enqueued for 3 packet transmission times, leading to three packets in a row arriving at input port A or hop 3. Suppose exactly the same situation happened for other red packet streams arriving at input ports B,C and D at hop 3. This would cause queueing of $i * (K-1) = 9$ packets at hop 3.