R2C2: A Network Stack for Rack-scale Computers

Paolo Costa Hitesh Ballani Kaveh Razavi* Ian Kash

Microsoft Research

ABSTRACT

Rack-scale computers, comprising a large number of microservers connected by a direct-connect topology, are expected to replace servers as the building block in data centers. We focus on the problem of routing and congestion control across the rack's network, and find that high path diversity in rack topologies, in combination with workload diversity across it, means that traditional solutions are inadequate.

We introduce R2C2, a network stack for rack-scale computers that provides flexible and efficient routing and congestion control. R2C2 leverages the fact that the scale of rack topologies allows for low-overhead broadcasting to ensure that all nodes in the rack are aware of all network flows. We thus achieve rate-based congestion control without any probing; each node independently determines the sending rate for its flows while respecting the provider's allocation policies. For routing, nodes dynamically choose the routing protocol for each flow in order to maximize overall utility. Through a prototype deployed across a rack emulation platform and a packet-level simulator, we show that R2C2 achieves very low queuing and high throughput for diverse and bursty workloads, and that routing flexibility can provide significant throughput gains.

CCS Concepts

•**Networks** → **Data center networks;** *Transport protocols; Cloud computing;*

Keywords

Rack-scale Computers; Congestion Control; Route Selection; Rack-scale Network Stack

*Work done during internship at Microsoft Research. Currently, PhD student at VU University Amsterdam.

SIGCOMM '15, August 17 - 21, 2015, London, United Kingdom

 ISBN 978-1-4503-3542-3/15/08…$15.00

DOI: http://dx.doi.org/10.1145/2785956.2787492

1. INTRODUCTION

While today's large-scale data centers such as those run by Amazon, Google, and Microsoft are built using commodity off-the-shelf servers, recently there has been an increasing trend towards server customization to reduce costs and improve performance [50, 54, 55, 58]. One such trend is the advent of "rack-scale computing". We use this term to refer to emerging architectures that propose servers or *rack-scale computers* comprising a large number of tightly integrated systems-on-chip, interconnected by a network fabric. This design enables thousands of cores per rack and provides high bandwidth for rack-scale applications. The consequent power, density and performance benefits means that racks are expected to replace individual servers as the basic building block of datacenters. Early examples of rack-scale computers include commercial (HP Moonshot [56], AMD SeaMicro [62], Boston Viridis [51], and Intel RSA [26, 59]) as well as research platforms [7, 9, 19, 34, 38].

A design choice that allows rack-scale computers to achieve high internal bandwidth and high density is to move away from a switched network fabric to a "distributed switch" architecture where each node functions as a small switch and forwards traffic from other nodes. This underlies many existing designs [19, 34, 38, 47, 51, 56, 59, 62], and results in a multi-hop direct-connect topology, with very high path diversity. This is a departure from today's data centers, which mostly use tree-like topologies. While direct-connect topologies have been used in high performance computing (HPC), the use of racks in multi-tenant datacenters means that network traffic is expected to be more diverse and unpredictable than in HPC clusters.

In this paper, we study two fundamental questions for the rack's network fabric: how should traffic be routed and how should the network be shared? The aforementioned peculiarities of rack environments pose challenges on both fronts. For *routing*, a one-size-fits-all approach is undesirable [1]. While there are many existing routing algorithms for direct-connect topologies, no single algorithm can achieve optimal throughput across all workloads (§2.2.1). For network sharing, existing *congestion control* approaches either do not cope with the high path diversity in racks (TCP family), or they are customized to specific workloads (HPC solutions [17, 18, 20]).

We present R2C2,[1] a network stack for rack-scale computers that provides flexible routing and congestion control. R2C2 achieves *global visibility*—each rack node knows all active flows—by broadcasting flow start and finish events across the rack. The scale of rack-scale computers (up to a few thousand nodes) allows for low overhead broadcasting. Given global visibility, each node independently computes the fair sending rates for its flows (§3.3). To account for temporary discrepancies in flow visibility, the rate computation leaves aside a small amount of bandwidth headroom. Such congestion control obviates any network probing or specialized queuing at rack nodes, yet it can accommodate high multi-pathing, and achieves both low network queuing and high utilization. Furthermore, it allows the provider to specify rich rate allocation policies, beyond flow-level policies. For routing, nodes locally determine how flows should be routed to optimize a provider-specified metric like aggregate or even tail rack throughput. Leveraging the global visibility and ensuring that nodes optimize for a global metric instead of selfish optimizations avoids any price of anarchy inefficiency [42] (§3.4).

At the data plane, both the rate allocation and route for a flow are enforced at the sender by rate limiting the flow's traffic (one rate limiter per flow) and encoding its network path into packet headers respectively. Intermediate rack nodes can thus simply forward packets along the path specified in their header, without requiring extra rate limiters or complex queuing mechanisms on path. By placing more functionality at the sender, this design enables a simple forwarding layer that is amenable to on-chip implementation.

Overall, this paper makes the following contributions:

- We describe a novel approach for rate-based congestion control that transforms the distributed network sharing problem into one of local rate computation.
- We describe a routing mechanism that allows for routing protocols to be chosen on a per-flow basis. We also present a greedy heuristic that rack nodes can use to locally determine the routing protocol for each flow in order to maximize a global utility metric.
- We develop a flexible emulation platform that enables accurate emulation of the network fabric in rack-scale computers. We use it to validate R2C2's design.

We implemented R2C2 as a user-space network stack atop our emulation platform. We use platform experiments to benchmark our implementation and cross-validate our packet-level simulator. Our simulation results show that R2C2 can achieve efficient network sharing for diverse network workloads: it achieves high throughput, fairness and low latency, and it imposes low broadcasting overhead. Finally, we show that routing flexibility and the dynamic selection process enables achieving higher performance that what would be possible using a single routing protocol for all flows.

[1]R2C2- Rack Routing and Congestion Control.

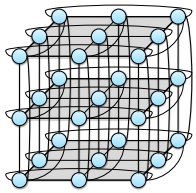


Figure 1: A 27-server (3x3x3) 3D torus. Each server has six neighbors.

2. BACKGROUND AND MOTIVATION

We begin by describing the factors underlying the emergence of rack-scale computers. We then delve into two very basic aspects of the network fabric inside a rack, *routing* and *congestion control*. By highlighting salient features of the network fabric, we argue why traditional solutions are insufficient in the rack environment.

2.1 Rack-scale computing

Rack-scale computers comprise 100s or even 1,000s of micro-servers that are connected by a network fabric. Their emergence is due to two hardware innovations. First, *System-on-chip (SoC)* integration combines cores, caches, and network interfaces in a single die. SoCs enable vendors to build *micro-servers*: extremely small server boards containing computation, memory, network interfaces, and sometimes flash storage. For instance, the Calxeda ECX-1000 SoC [52] hosts four ARM cores, a memory controller, a SATA interface, and a low-radix switch onto a single die.

Second, *fabric integration* means that these micro-servers (or *nodes*) can be connected through a high-bandwidth low-latency network. This is typically done through a "distributed switch" architecture; rather than connecting all nodes to a single ToR switch like in today's racks, each node is connected to a small subset of other nodes via point-to-point links. These links offer high bandwidth (10–100 Gbps) and low per-hop latency (100-500 ns). Any topology that has a small number of links per node can be used; 2D and 3D torus like the one in Figure 1 are popular choices adapted from super-computing architectures. These topologies are often referred to as *multi-hop direct-connect topologies* because packets typically have to travel multiple hops before reaching the destination and nodes are responsible for forwarding packets at intermediate hops (typically using an integrated switching element on the SoC).

Early examples of rack-scale computers have appeared on the market. For example, HP's Moonshot [56] is a 4.3 rack-units chassis with 45 8-core Intel Atom SoCs and 1.4 TB of RAM in a 3D torus topology. The AMD SeaMicro 15000-OP [62] stacks 512 cores and 4 TB of RAM within 10 rack-units using a 3D torus network fabric with a bisection bandwidth of 1.28 Tbps. Intel's proposed Rack-scale Architecture [53, 59] combines SoC and fabric integration with silicon photonics, which support link bandwidths of 100 Gbps and higher. Rack-scale computer designs have also been proposed by the research community such as the Catapult [38], Firebox [7], Pelican [9], and soNUMA [19, 34] platforms.

Workload	Random Packet Spraying	Destination Tag routing	VLB	WLB
Nearest neighbor	4	4	0.5	2.33
Uniform	1	1	0.5	0.76
Bit complement	0.4	0.5	0.5	0.42
Transpose	0.54	0.25	0.5	0.57
Tornado	0.33	0.33	0.5	0.53
Worst-case	0.21	0.25	0.5	0.31

Figure 2: Throughput, as fraction of network bisection capacity, of four routing algorithms on an 8-ary 2-cube on size traffic patterns (table from [20]). No single routing algorithm can achieve optimal throughput across all workloads.

2.2 Rack-scale networking

Rack-scale networking combines the topology of HPC systems with the workloads of traditional datacenters. Like HPC clusters, racks have a multi-hop direct-connect topology with very high path diversity. This is in contrast to tree-like topologies common in datacenters. As for the workload, we expect racks to be used in multi-tenant environments with different applications generating diverse network workloads, unlike the homogeneous and relatively stable workloads typically observed in HPC.

In the following sections, we focus on two fundamental questions for rack-scale networking: how should traffic be routed and how should the network be shared? We describe the shortcomings of existing solutions, and distill design goals for a rack-scale's network stack.

2.2.1 Rack routing

Routing across direct-connect topologies has been extensively studied in the scientific computing and HPC literature [20]. Existing routing algorithms can be broadly classified into two categories. Minimal routing algorithms like randomized packet spraying [22] and destination tag routing [20] route packets only along shortest paths. Non-minimal algorithms like Valiant Load Balancing (VLB) [45] and Weighted Load Balancing (WLB) [44] do not restrict themselves to shortest paths.

High path diversity in direct-connect topologies means that no single routing algorithm can achieve optimal throughput across all workloads. Minimal routing ensures low propagation delay but at the expense of load imbalance across network links which, in turn, results in poor worst-case performance. On the other hand, VLB transforms any input traffic matrix into a uniform random metric by routing packets through randomly chosen waypoints. This ensures guaranteed worst-case throughput across all workloads but hurts average-case throughput, especially for workloads with locality. WLB lies between these extremes; it considers non-minimal paths for load-balancing but biases the path selection in proportion to the path length.

The table in Figure 2 summarizes these arguments by showing the throughput of these routing algorithms for five traffic patterns and their worst-case throughput (the worst-case traffic patterns for each algorithm are different). While VLB has the same performance across all traffic patterns, other algorithms perform better for specific workloads. By contrast, tree-like topologies in today's datacenters have a few orders of magnitude less multi-pathing, so minimal routing algorithms like packet spraying are sufficient [22].

Overall, for multi-tenant datacenters where the network traffic pattern is not known a priori and expected to change over time, a one size fits all approach is undesirable [1]. Thus, our first design goal is:

G1 *Routing flexibility.* The network stack should allow for different routing protocols to be used simultaneously, chosen on a per-application or even on a per-flow basis. Datacenter operators can leverage this knob to optimize metrics like the rack's aggregate throughput.

2.2.2 Rack network sharing

High path diversity, combined with the resource limited nature of micro-servers, also means that congestion control protocols used in today's datacenters are not suited for rack-scale computers. For example, a flow using minimal routing is routed along *all* shortest paths between its source and destination. This means that even in a small 216-node rack with 3D torus topology, an average flow has a 1,680 paths. Furthermore, the number of paths increases exponentially with the topology size. And with non-minimal routing, the path diversity can be nearly unbounded [20].

The TCP family of protocols, including recent proposals targeted towards datacenters [2–4], only uses a single path and imposes high processing overhead [34]. Even multi-path extensions like MPTCP [41] only consider few tens of paths. This is roughly two-three orders of magnitude smaller than the number of paths available here. Furthermore, per-flow fairness, as provided by TCP and its variants, is inadequate for datacenter settings. Instead, operators need to enforce richer policies like deadline-based fairness for user-facing applications [28, 48] and per-tenant guarantees [10, 11, 30].

In recent work, Fastpass [36] shows the feasibility of centralized congestion control in traditional datacenters. While the scale of rack-scale computers is amenable to a centralized design, this would introduce significant communication overhead, as we show in Section 5.2. Further, high path diversity in racks makes computing max-min fair rate allocations much harder than in traditional topologies (§3.3). In RascNet [16], we sketched a preliminary congestion control design for racks that works atop VLB routing. However, this contravenes the routing flexibility requirement. In Section 5.2, we show that taking advantage of routing flexibility provides significant performance gains.

At the other end of the spectrum, HPC platforms and on-chip networks (NoC) systems often use congestion control mechanisms customized to the underlying topology and the expected workload [17, 18, 35]. Such mechanisms do not work for general workloads. We also note that ideal network sharing can be achieved through per-flow queues at each rack node coupled with back-pressure notifications when a node's queues start filling up. Apart from increased forwarding complexity, this massively increases the buffering requirements at rack nodes.

Thus, the design goals for rack network sharing are:

G2 *Accommodate high multi-pathing.* Congestion control

has to cope with high multi-pathing in rack topologies.

G3 *Low network queuing.* While a standard goal for network design, this is particularly important here because micro-servers have limited buffers and the network carries traffic that is very latency sensitive.

G4 *Allocation flexibility.* Datacenter operators should be able to specify different rate allocation policies that encapsulate varying notions of fairness.

3. DESIGN

We present R2C2, a network stack for rack-scale computers. The key insight behind R2C2's design is that while rack topologies pose many challenges, they also present an opportunity—it is possible to efficiently broadcast information across the rack. We use 16-byte broadcast packets (§4.2); with a 512-node rack, each broadcast results in 8 KB of total traffic, aggregated across all rack links. By broadcasting flow start and termination events, we ensure every node knows the rack's global traffic matrix. R2C2's leverages such *global visibility* to implement flexible congestion control and routing.

Determining the rates and routes for flows across a multi-path topology can be mapped to the multi-commodity flow (MCF) problem with splittable flows. Several papers propose polynomial time algorithms for this problem with different optimization objectives such as max-min fairness [33] and maximizing total throughput [8]. Many of these algorithms are designed for offline operation; thus they are computationally intensive and have a high running time. Rack nodes, however, have limited compute and buffering, so at least the congestion needs to be controlled at a fine-grained timescale. On the other hand, online MCF algorithms are tightly tied to a specific optimization metric.

R2C2's control plane decouples congestion control and routing, and tackles them at different timescales. For *congestion control*, each node uses global visibility and knowledge of the underlying topology to locally compute the fair sending rate for its traffic (§3.3). This design avoids any network probing and does not require any specialized queuing support at the rack nodes. The rate computation algorithm is fast, ensures low network queuing, and allows for different rate allocation policies. For *routing*, nodes locally determine the routing protocol for each flow that will maximize a provider-chosen global utility metric (§3.4).

At the data-plane, R2C2 uses source routing to enable per-flow routing protocols. This involves three mechanisms: (i). the node sending a flow determines the path for each packet based on the flow's routing protocol and encodes this path in the packet header, (ii). the sender also enforces a flow's rate allocation, (iii). intermediate rack nodes simply forward packets along the path specified in their header.

Overall, our design choices are guided by the expected size of rack-scale computers. Their scale ensures both the network overhead of broadcast traffic and the processing overhead of rate computation is acceptable, even for very bursty workloads. The scale also means that a packet's path can be encoded compactly, allowing for source routing. The data-plane design intentionally places more functionality at the sender that only needs to rate limit its own flows. This can be implemented in software or hardware [39]. Intermediate rack nodes have a simple forwarding layer that is amenable to on-chip implementation; it does not require any additional rate limiting or complex queuing mechanisms.

To give an overview of R2C2's operation, we begin by focusing on the life of a single flow. We then describe R2C2's data- and control-plane mechanisms in detail.

3.1 The life of a flow

When a flow starts, its sender broadcasts information about the new flow, including the routing algorithm the flow is using and its rate allocation parameters (e.g., the flow's weight). Each rack node stores this information to create a local view of the global traffic matrix. Given this traffic matrix and the rack's topology, the sender computes the flow's fair allocation and rate limits it accordingly. To account for temporary discrepancies between the perceived and actual traffic matrix, R2C2 relies on bandwidth headroom; during rate computation, we simply subtract the headroom from each link's capacity.

For every new packet, the source encodes the packet path in its header, and the packets are source routed to their destination. When a broadcast packet (e.g., due to a flow starting or finishing) is received, the sender recomputes the rate for all its *own* flows. When the flow finishes, other rack nodes are informed by broadcasting this event. Nodes also periodically check whether the overall utility would improve if some of the flows were routed using a different protocol. If a significant improvement is possible, their routing protocols are changed and this information is broadcasted.

3.2 Broadcast

R2C2's design builds upon a low overhead broadcast primitive. For broadcasting packets, we create a per-source broadcast tree atop the rack's network topology. This can be done while optimizing various goals; we optimize the broadcast time, i.e., we minimize the maximum number of network hops within which all rack nodes receive a copy of the broadcast packet.

To achieve this goal, we determine the *shortest-path tree* for each rack node. Given a graph representing the rack's topology, a shortest-path tree rooted at source node s is a spanning tree T of the graph such that the length of the path from s to any node in T is the shortest distance from s to the node in the graph.[2] Since all network links inside the rack have the same capacity, finding shortest-path trees for the rack is akin to finding shortest-path trees for an unweighted graph, and can be done through a breadth-first traversal of the graph [14].

For a rack, we enumerate multiple broadcast trees for each source by traversing the rack's topology in a breadth-first fashion. Given this, we construct a broadcast forwarding information base (FIB) for each rack node. A look-up in this FIB is indexed by a two-tuple, <`src-address, tree-`

[2]There can be multiple shortest-path trees for a given source.

id>, comprising the address of the source node and an identifier for the broadcast tree, and yields the set of next-hop nodes the broadcast packet should be forwarded to.

When a node has to send a broadcast packet, it chooses one of its broadcast trees for the packet to be routed along. This selection is done to load balance the broadcasting overhead, and allows the sender to account for link and node failures. The sender inserts its address and the identifier for the chosen broadcast tree into the header of the broadcast packet. The packet is then routed by other nodes by consulting their broadcast FIB.

Broadcast overhead. In Section 4.2, we describe how flow information is encoded into a 16-byte broadcast packet. Each broadcast tree for a rack with n nodes comprises $n-1$ edges. Thus, for a typical rack with 512 nodes, a single broadcast results in a total of 511*16 $\approx$ 8 KB on the wire. In the worst-case scenario of flows between all pairs of nodes ($\approx$262K flows), the resulting broadcast traffic per link would be 681 KB.

An obvious concern is that in many datacenter workloads, most flows are only a few packets long. For example, in a typical data-mining workload [25], 80% of flows are less then 10KB. The average path length for a flow in a 512-node 3D torus is 6 hops, so a 10 KB flow will, on average, result in 60 KB being transmitted on the wire. Thus, the relative overhead of broadcasting the start and finish events for such small flows is 26.66% (13.33% for each event). Fortunately, small flows only carry a small fraction of the bytes in today's datacenters. For example, traces from a data analytics cluster show that the 95% of all bytes originate from less than 4% of flows [25]. In Section 5.2, we analyze the fraction of network bandwidth used for broadcast traffic as a function of the bytes carried by small flows. When 5% of the bytes are carried by small flows, the fraction of the network capacity used for broadcasting flow information is only 1.3%.

Failures. Broadcast packets can be corrupted across the network, or lost due to drops and failures. To detect corruption, we rely on a packet checksum. To detect drops due to queue overflows at intermediate nodes, the node dropping a broadcast packet informs the sender who can then re-transmit. To detect link and node failures, we rely on a topology discovery mechanism that is required by the routing protocols anyway. Upon detecting a failure, nodes broadcast information about all their ongoing flows. This is reasonable because, given the scale of rack-scale computers, we expect node and link failures to be infrequent. For example, measurements across HPC systems have shown a failure rate of around 0.3 faults per year per CPU [43]. For a 512-node rack with four CPUs per node, this means less than two failures a day.

3.3 Rack congestion control

To ensure the network is not congested and flows achieve rates in accordance to the operator's allocation policy, we require senders to compute the rate allocations for their traffic and enforce them. The basic idea behind our approach is that given knowledge of the allocation policy, the network topology, all active flows, and their current routing protocol, each node can independently determine the load on each network link and hence, the fair sending rate for its flows. Thus, we transform the distributed congestion control problem into one of local rate calculation. While the rack's topology is relatively static, the set of active flows can change rapidly. We begin with a strawman design which assumes that nodes are aware of the rack's current traffic matrix.

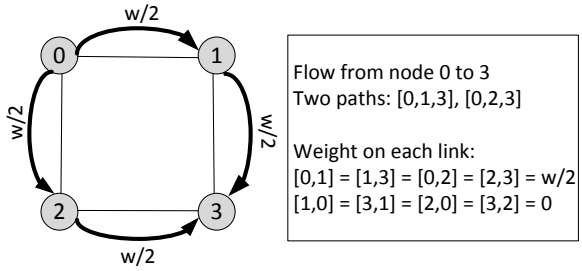


Figure 3: A flow from node 0 to 3, with weight w, being routed using randomized packet spraying.

The key challenge in computing rate allocations is accommodating high multi-pathing. Flows are routed across a very high number of paths, thousands or more, which poses a computational burden. For example, consider a provider who wants to allocate the network fairly. Max-min fairness is well studied in the context of single-path routing but is harder to reason about in multi-path settings. Max-min Programming (MP) [40] is a centralized algorithm that uses a linear program to compute max-min fair allocations across general networks. However, using the MP algorithm in our setting would result in a linear program with an exponential complexity solution—each flow takes many paths, each path would be represented by a separate variable.

To make rate computation tractable, we leverage the simple insight that a flow's routing protocol dictates its relative rate across its paths. This holds for the routing protocols we studied. For instance, consider the example in Figure 3, in which a flow from 0 to 3 is routed using random packet spraying [22] across a 2x2 mesh topology. There are two shortest paths between the source and the destination, $0 \rightarrow 1 \rightarrow 3$ and $0 \rightarrow 2 \rightarrow 3$, which are chosen randomly on a per-packet basis and hence, are used equally. Therefore, the total rate allocated to the flow should be evenly divided across these two paths.

Overall, the key observation that a flow's routing protocol dictates its relative rate across multiple paths allows us to compute rate allocations at the flow-level, irrespective of the number of paths each flow is routed along.

3.3.1 Congestion control: a strawman design

R2C2's congestion control involves each node independently computing the max-min fair rate for each of its own flows. This comprises two steps. First, we use information about a flow's routing protocol to determine the relative rate of the flow across the paths it is using and, hence, along each network link it is using. We explain this with an example. To achieve per-flow fairness, each flow is assigned the same allocation weight. Given the flow's source, destination and

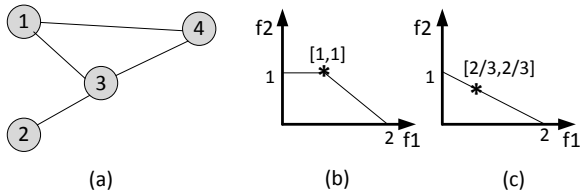


Figure 4: Flow $f1$ from node 1 to 4, and $f2$ from node 2 to 4. Respecting the relative rates dictated by the routing protocol changes the feasible set of rates from (b) to (c). The asterix denotes the max-min fair allocation.

routing protocol, we can determine its weight along any network link. For instance, the flow in Figure 3 is using both minimal paths equally, so its weight on each link that it is using is $\frac{w}{2}$. The flow's weight on each link it is not using (e.g., the link $1 \rightarrow 0$ in the figure) is zero. As another example, assume the flow was using WLB routing and using the two paths, $0 \rightarrow 1 \rightarrow 3$ and $0 \rightarrow 2 \rightarrow 3$, in the ratio 1:2. Then the flow's weight at links $0 \rightarrow 1$ and $1 \rightarrow 3$ is $\frac{w}{3}$, and its weight at links $0 \rightarrow 2$ and $2 \rightarrow 3$ is $\frac{2w}{3}$. We can repeat the above process for all flows to determine per-flow weights at all network links.

Second, given this setup, we compute max-min fair allocations through a weighted version of the water-filling algorithm [12] that we summarize here: the rates for all flows are increased at the same pace, until one or more links become saturated. Note that a flow's rate across a given link is a product of the flow's total rate and its weight across the link. This saturated link(s) is the bottleneck, and its capacity dictates the rates for all flows through it. The rates for these flows are frozen and they marked as allocated. The algorithm continues till all flows have been allocated. The complexity of this algorithm is $O(NL+N^2)$, where N is the number of flows and L is the number of links.

Benefits. This strawman design has a few advantages. First, it avoids the need to probe the network and induce congestion signals like packets drops and queuing to infer a flow's sending rate. High path diversity in rack-scale computers makes the design of an efficient congestion probing mechanism particularly challenging. Second, it obviates the need for any specialized queuing at the intermediate rack nodes.

Drawbacks. The computational tractability of the algorithm comes at the expense of possible network under-utilization. Consider the example scenario in Figure 4(a) (from [40]) with links of capacity one and two flows: flow $f1$ from node 1 to 4, and $f2$ from node 2 to 4. We assume the routes for these flows are given; $f1$ uses two paths ($1 \rightarrow 4$ and $1 \rightarrow 3 \rightarrow 4$) with equal probability while $f2$ uses one path ($2 \rightarrow 3 \rightarrow 4$). The set of feasible rates for these flows are shown in Figure 4(b), and the ideal max-min fair allocation is $\{1, 1\}$, i.e., each flow sends at a unit rate. This means flow $f1$ only uses the path $1 \rightarrow 4$. In contrast, R2C2's congestion control ensures that flow $f1$'s rate across its two paths is equal (as dictated by its routing protocol). The restricted set of feasible rates is shown in Figure 4(c). Our algorithm converges to the max-min fair rate in this set, $\{\frac{2}{3}, \frac{2}{3}\}$.

In summary, respecting the relative rates dictated by the routing protocol can result in network under-utilization. However, as described in Section 3.4, R2C2 also adapts the routing of all long running flows to alleviate this. For our example, flow $f1$'s routing would be changed so it only uses the path $1 \rightarrow 4$.

3.3.2 Congestion control extensions

The simple design described above makes several simplifying assumptions. Below we relax these assumptions.

New flows. The strawman design assumes that nodes are aware of the rack's current traffic matrix. While a flow startup event is broadcasted to all other nodes, the sender starts transmitting packets immediately. Thus, there is a small time period when not all nodes are aware of a new flow that has already started. To account for such temporary discrepancy between the actual traffic matrix and what is perceived by rack nodes, we rely on bandwidth headroom. During rate computation, we subtract this headroom from the capacity of network links. Typical datacenter traffic patterns involve many short flows that only contribute a small fraction of the bytes across the network. This indicates that a small spare capacity should absorb such short-lived flows while our congestion control protocol ensures that medium-to-long flows do not congest the network. In our experiments detailed in Section 5, a 5% headroom is sufficient even with bursty traffic patterns.

We note that R2C2 does not need a new data-plane mechanism (like phantom queues at switches [3]) to achieve the bandwidth headroom. Instead, the headroom is incorporated into the rate computation done at the control plane.

Host-limited flows. The strawman design assumes all flows are network limited. In practice, however, flows can be bottlenecked at their end points, e.g., at the application itself or inside the OS. To ensure high network utilization, any bandwidth unused by such host-limited flows should be allocated to flows that can use it. To account for host-limited flows, we estimate flow *demand*—the maximum rate at which it can actually send traffic—at the sender. Given a flow's demand estimate, the rate assigned to it is the minimum between its fair share and its demand. Whenever a flow's demand drops below it current rate allocation, the sender broadcasts this information. This allows all rack nodes to compute allocations in a demand-aware fashion.

Demand estimation is based on the observation that a flow sending at a rate higher than what it is currently allocated will suffer queuing at the sender. The sending node uses this queuing information to estimate the flow's demand. Specifically, flow demand is estimated periodically, and is the sum of the rate it is currently allocated and the amount of queuing it observes. Thus,

$$d[i+1] = r[i] + q[i]/T \quad (1)$$

where $d[i+1]$ is the estimated demand for the next period, T is the estimation period while $r[i]$ and $q[i]$, respectively, are the rate allocation and queuing observed by the flow in the current period. To smooth out any noisy observations, we use an exponentially weighted moving average of the esti-

mated demand. This estimation applies to open-loop workloads. For closed-loop workloads, demand can similarly be estimated using queuing information [6].

Beyond per-flow fairness. The strawman design provides per-flow fairness. However, R2C2 can accommodate richer allocation policies by allowing the operator to specify a flow's *weight* and *priority*. Many recently proposed high-level fairness policies such as deadline-based [46] or tenant-based [37], can be mapped onto these two primitives, similar to pFabric [4]. A flow's weight and priority are included in the broadcast packet announcing the flow start, so all nodes can use them during the rate computation.

The rate allocation algorithm accounts for flow priorities and weights as follows. Nodes invoke the allocation algorithm over multiple rounds, one for each priority level. At each round, flows belonging to the corresponding priority level are allocated any remaining capacity in a weighted fashion, i.e., instead of using uniform weights (as in Section 3.3.1), each flow is associated with its own weight.

Periodic rate computation. The strawman design recomputes flow rates at every flow event. Even at rack-scale, this will impose too much computation overhead. To amortize the recomputation cost, we opted for a batch-based design in which rates are recomputed periodically. In Section 5, we show that for realistic workloads a recomputation interval in the range of 500 μs-1 ms is sufficient to ensure high utilization and low queuing while introducing a CPU overhead of less than 8% at the 99^{th} percentile (median value is 1.7%).

While this design was motivated by the need to reduce the computation cost, it also naturally filters out very short-lived flows, which would be pointless to rate-limit.

3.4 Selecting routing protocols

For short flows, a minimal routing protocol can improve the flow completion time. Thus, new flows start with minimal routing. As flows age, their routing can be adapted based on the rack's traffic matrix. R2C2 periodically selects the routing protocol for each long flow to maximize a global utility metric specified by the datacenter operator. The adaption is done every few seconds or minutes. Example utility metrics include the rack's aggregate throughput or the tail throughput, as measured across tenants or even across jobs and application tasks [15, 23]. For ease of exposition, here we focus on maximizing the aggregate throughput.

Dynamic selection of routing protocols is challenging because we want to leverage the flexibility of choosing a different routing protocol for *each* flow. This results in a combinatorial number of flow and routing protocol combinations to be evaluated, which makes exhaustive search intractable. Further, the search landscape originating from the utility functions that we considered typically exhibits several local maxima. Therefore, simple greedy heuristics (e.g., hill-climbing) are not effective and more complex global search heuristics must be adopted.

Initially, we considered techniques such as *log linear learning* [5] and *simulated annealing* [13]. However, we found them very sensitive to parameter tuning and workload characteristics. Thus, we opted for *genetic algorithms* [27], a search heuristic that emulates the natural selection and evolution. We found it a good fit for our scenario because it has relatively few tuning parameters and our problem can be naturally encoded as bit strings, where one or more bits are used to identify the routing protocol assigned to a given flow.

The heuristic works as follows. Initially, we generate a population of flow and routing protocol combinations (*genotypes*) that contains the current routing allocation and other randomly-generated ones. For each genotype, we compute the rack's aggregate throughput (*fitness*) using the rate computation mechanism described in Section 3.3 and rank them accordingly. We then generate a new generation that contains the top genotypes of the current population and other genotypes obtained by recombining (*crossover*) and *mutating* existing genotypes. The process is repeated until the time expires or until there has been no improvement over a number of generations. Once the search is over, the server generates broadcast packets containing the new assignment for each flow. Our current implementation uses four bytes for the flow identifier and one byte for the routing protocol per each flow (§4.2). This means that up to 300 {flow, routing protocol} pairs can be advertised using a single 1,500-byte packet. Since this process only applies to long flows, we consider the overhead negligible.

For simplicity, in our prototype, a single node is responsible for periodically performing the routing selection process. In practice, we expect this operation be decentralized with each node in turn executing it. Since nodes optimize a global utility metric, instead of selfishly optimizing for local performance, this design does not suffer from any price of anarchy inefficiency [42]. The next node to run the routing selection heuristic can be chosen randomly using a probability distribution centered around the average adaptation period or using a deterministic token-based scheme whereby nodes are selected serially in a round robin fashion.

3.5 Rack data-plane

R2C2 places most data-plane functionality at the source, resulting in simplified forwarding at intermediate rack nodes. This design is particularly suited to direct-connect topologies as each packet is forwarded through many nodes.

For each packet, its sender uses the corresponding flow's routing protocol to determine the packet's path. The path is then encoded into the packet's header. Further, each flow is associated with a token bucket that rate-limits the flow to its current rate allocation. Intermediate nodes forward packets using source routing. The packet header contains a route index field indicating the index of the next-hop in the packet path. Thus, every intermediate node simply forwards the packet to the indicated next-hop after incrementing the route index in the packet header.

4. IMPLEMENTATION

We have developed a flexible and efficient emulation platform for rack-scale computers, and implemented R2C2 as a user-space network stack atop this platform.

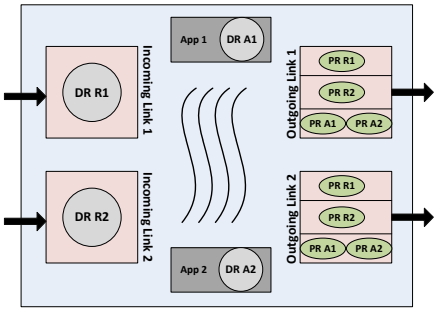


Figure 5: An example of a Maze server emulating a rack node with two incoming and two outgoing links. The emulated node has two applications.

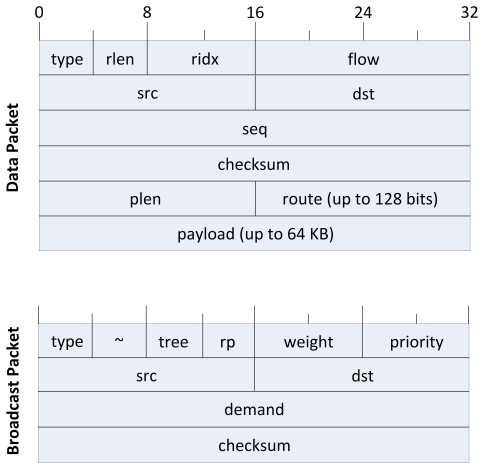


Figure 6: The format of data and broadcast packets.

4.1 Rack emulation platform

Rack-scale computers are currently difficult to acquire. Further, they come pre-configured with static network topologies and routing protocols, making it difficult to ensure that a proposed idea that works on one rack-scale computer will work on another. To address these issues, we implemented *Maze* [60], a cluster-based network emulation platform focusing on rack-scale fabrics. Maze runs on a cluster of servers connected by a high-bandwidth RDMA-based switched network. It emulates a rack's network fabric as a virtual network atop the switched network. Below we describe Maze's key properties and summarize its operation.

Maze provides three key properties: (i) It is *configurable*. It can map any virtual network topology (e.g., tree, mesh, torus, etc.) onto the underlying cluster. It can also support multiple routing strategies and transport protocols. (ii) It offers *high-performance* emulation, with the ability to emulate high capacity network links. Through micro-benchmarks, we find that it can provide up to 38 Gbps on a 40 Gbps link using 8 KB packets, and a latency of 3 μs per hop using small packets. (iii) It achieves *high-fidelity*. Our micro-benchmarks show that Maze can faithfully emulate many virtual links on the same physical network link.

Figure 5 depicts the operation of a Maze server emulating two incoming and two outgoing links. To achieve the properties mentioned above, Maze uses three main techniques: 1) It uses RDMA to transfers packets, 2) uses zero-copy forwarding, and 3) implements flow rate control.

Packet transfer. To provide high-performance yet configurability and ease of experimentation, Maze allows new routing and transport protocols to be implemented in user-space. It uses RDMA writes from the senders to data ring buffers (DR) on receivers' memory (e.g., DR R1 in Figure 5), similar to recent RDMA-based key-value stores [24, 31]. In Maze, incoming links, as well as applications (running emulated nodes), register memory to the RDMA NIC, which is accessed during RDMA writes. Pointer rings (PR) reference the registered memory and they are used in the outgoing links (e.g., PR R1) during RDMA writes.

Forwarding. An outgoing link on a Maze server constitutes a connection to another Maze server (i.e., RDMA queue pair [61]) and a number of pointer rings. To achieve zero-copy forwarding, for each packet, depending on its routing information, we pass the packet pointer that references the incoming ring buffer to the associated pointer ring on the selected outgoing link (e.g., PR R2 on outgoing link 1 sends packets of DR R2 on incoming link 2). Once the packet is sent, we zero the memory of the forwarded packet to make space for new packets in the respective receive ring buffer. We send application packets in the same fashion (e.g., using DR A1 and its respective pointer rings).

Rate control. Maze provides different pointer rings for different flows. This allows for implementation of both back-pressure based and rate-based congestion control approaches. Transport protocols can use Maze's rate limiters to adjust the rate at which data pointers from a flow pointer are inserted on an application's pointer rings belonging to outgoing links.

4.2 R2C2 implementation

We implemented R2C2 as a user-mode network stack in Maze. For routing, we have implemented random packet spraying [22], destination-tag routing [20] and VLB routing [45]. For congestion control, we have implemented our rate-based congestion control protocol. This leverages Maze's rate control functionality. Overall, our complete prototype (including Maze) consists of 9,836 lines of C++ and 967 lines of control scripts.

Rate computation. We implemented an efficient variant of the water-filling algorithm described in Section 3.3.1. A key challenge was how to efficiently compute the link weights for each flow. We solved this by pre-computing on each node the list of link weights for each {routing protocol, destination} pair. Assuming a 512-node rack, the memory footprint per routing protocol is less than 6 MB, i.e., 511 destinations times the number of links (6·512) times 4 bytes for the weight.

Rate limiters. As explained in Section 3.3.1, the R2C2 congestion control respects the relative rates dictated by the routing protocol. Beside reducing the computation over-

head, this also implies that only one rate limiter per flow is needed as opposed to one rate limiter per each of the paths traversed by the flow. Further, flows are only rate-limited at the source and not at the intermediate nodes. Therefore, the number of rate limiters needed by each node is equal to the number of flows that it generates. In our prototype, we rely on software rate limiters, which can achieve very fine-grained rate limiting [29]. Furthermore, we believe that these requirements are well within the reach of today's NICs, which typically support 8-128 hardware rate limiters [39].

Packet formats. Figure 6 shows the format used for data and broadcast packets. Data packets are variable sized, while broadcast packets have a fixed size (16 bytes). The packet type is defined in the *type* field.

The header of a data packet contains length of the route that the packet travels (*rlen*), an index into the packet's route (*ridx*), the flow identifier (*flow*), source (*src*), destination (*dst*), the sequence number (*seq*), the packet checksum (*checksum*), length of the payload (*plen*), the packet's route (*route*), and finally the payload.

The size of endpoints allows for up to 65,536 nodes. Further, the *route* field could be up to 128 bits. We use 3 bits for each hop to select the forwarding link (i.e., we assume at most eight links per node) and increase *ridx* every time we forward a packet. This allows us to store routes with up to 42 hops, which is sufficient for current rack-scale computers and even non-minimal routing strategies.

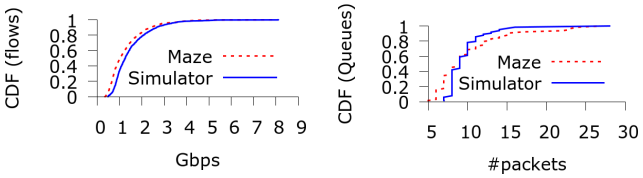
Apart from source, destination and the packet checksum, broadcast packets include the weight of the flow (*weight*), flow's priority (*priority*), the demand (*demand*) in Kbps between the nodes (up to 4 Tbps), the broadcast spanning tree id (*tree*), and the currently used routing strategy between the two nodes (*rp*).

5. EVALUATION

In our evaluation, we answer the following questions: (i). how effective is R2C2's congestion control mechanism in achieving high throughput and low queuing?, (ii). what is the traffic overhead introduced by broadcast?, (iii). what is the cost of rate computation?, and finally (iv). what are the benefits of supporting per-flow routing selection?

We adopted the following methodology. First, we use R2C2's implementation atop Maze to empirically verify the feasibility and performance of our design, to quantify the computation overhead, and to cross-validate our packet-level simulator. Then, we use the simulator to investigate the performance of R2C2 at scale and under different workloads.

Our results indicate that R2C2 achieves high throughput and fairness while only requiring small queues and the overhead imposed by broadcast is negligible. The computation cost depends on the frequency at which rates are recomputed. However, for realistic workloads, this cost is reasonable. Finally, by enabling individual flows to use different routing protocols and by dynamically selecting among them based on the observed workload, R2C2 achieves higher performance than what would be possible using only a single routing protocol for all flows.



(a) Flow throughput. (b) Queue occupancy.

Figure 7: Cross-validation of the flow throughput and maximum queue occupancy between Maze and the simulator using a 4x4 2D torus topology.

5.1 Emulation results

We deploy our Maze platform on a 16-server RDMA cluster. Each server is equipped with two 2.4 GHz Intel Xeon E5-2665 CPUs and 24 GB of memory. The servers are connected using Quad Data Rate (QDR) InfiniBand. We emulated a 4x4 2D torus virtual topology with a bandwidth of 5 Gbps per virtual link.

We generate a synthetic workload, comprising 1,000 flows of 10 MB each. We assume Poisson flow arrivals with a mean inter-arrival time of 1 ms. We use the random packet spraying routing protocol. We measure the throughput and the *maximum* occupancy experienced by each queue throughout the entire experiment. We then repeated the same experiment in the simulator, using the same topology and workload. Results in Figure 7 show that our packet-level simulator exhibits high accuracy, both in terms of flow throughput and queuing occupancy. These cross-validation results improve our confidence in the large-scale simulation experiments presented in Section 5.2.

Computation overhead. Next, we evaluate the computation overhead introduced by R2C2 when recomputing flow rates. To show the behavior at scale, we ran the same workload in the simulator using a 512-node 3D Torus and we recorded the flow arrival and departure events at each node. To account for the increase in scale, we reduce the flow inter-arrival time to 1 us. We then replayed these traces in Maze and measured the execution time of the rate recomputation. We ran this benchmark on two different CPU cores: a 2.4GHz Intel Xeon E5-2665 and a 1.66 GHz Intel Atom D510 [57]. The first one is representative of today's data center servers. The latter, instead, is a first-generation (2009) low-power CPU architecture. This could be used, for instance, as a cheap, dedicated core on each SoC to handle the execution of R2C2.

Figure 8 plots the 99^{th} percentile of the CPU overhead for different values of the recomputation interval ρ. We compute the overhead by dividing the time taken to recompute the rates by the value of ρ. This means that if the overhead is higher than 100% (the horizontal line in the chart), the recomputation would not finish in time and, hence, the interval is not feasible. As discussed in Section 3.3.2, we use a batch-based design in which we only consider the flows that last more than one interval. Therefore, the longer the interval, the lower the number of flows considered, and this explains why high values of ρ exhibit lower overhead. For example, for ρ=500 us, the median overhead on the Intel Xeon is

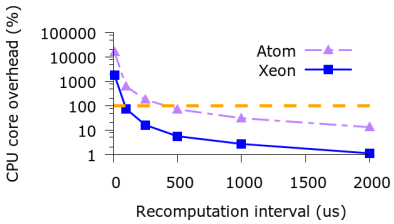


Figure 8: The 99^{th} percentile of the CPU overhead on one 2.4 GHz Intel Xeon E5-2665 core and on a 1.66 GHz Intel Atom D510 core when simulating a 512-node workload with 1 us flow inter-arrival time.

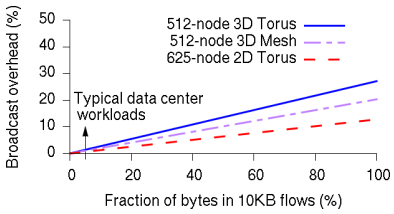


Figure 9: Network capacity used for broadcasting grows linearly with the fraction of bytes carried by small flows. The experiment involves uniform traffic with shortest path routing using 10KB small flows and 35MB long flows.

1.7% (99^{th} percentile is 7.9%) and 33.5% on the Intel Atom (99^{th} percentile 71.4%). For low values of ρ, the overhead becomes much higher, e.g., for ρ=100 us the 99^{th} percentile is 73.9% on the Intel Xeon (median is 17.6%) while it is infeasible for the Intel Atom (the normalized CPU usage is higher than 100%). However, as we show in Section 5.2, for the workloads we consider, a value of ρ=500 us is sufficient to achieve good performance.

Broadcast overhead. Finally, we analyzed the overhead of packet broadcasts on flow arrivals and departures. We consider the size of the broadcast and data packets in Maze to analytically project the results for a 512-node 3D torus. Assuming 10 KB flows (§3.2), the average bandwidth overhead is 26.66%. For 10 MB flows, instead, the overhead would just be 0.026%.[3] Thus, the overall broadcasting overhead depends on the fraction of bytes carried by small flows.

Figure 9 shows that the fraction of network capacity used for broadcast traffic grows linearly with the fraction of bytes carried by small flows. In typical datacenter workloads, small flows carry a small fraction of bytes; 95% of all bytes are in the 3.6% flows larger than 35 MB [25]. For such a workload, only 1.3% of the network capacity is used for broadcasting. The figure also shows that for topologies with a greater diameter like a 3D Mesh and a 2D Torus, the broadcast overhead is lower. This is because the average flow traverses more hops, so the relative overhead of broadcasting its arrival and departure is lower.

5.2 Simulation results

In our simulation experiments, we consider a 512-node 3D torus. This is the same size and topology as the AMD SeaMicro 15000-OP. We assume a link bandwidth of 10 Gbps and a per-link latency of 100 ns. We leave 5% headroom and, except where otherwise noted, we use a recomputation interval of 500 us. In our experiments, we use a synthetic workload modeled after traffic patterns observed in production data centers [2, 4, 25]. Flow's source and destination are randomly chosen following a uniform distribution. The flow sizes are generated from a Pareto distribution with shape parameter 1.05 and mean 100 KB [3]. This generates a heavy-tailed workload where 95% of the flows are less than 100 KB, as is commonly observed in data centers. We assume Poisson flow arrivals and we consider flow inter-arrival times varying from 1 us to 100 us. To stress our system, we also consider an extreme flow inter-arrival time of 100 ns, which corresponds to a workload with 10^{10} flows/s with a peak of 2,241 simultaneous flows. This is up to two orders of magnitude lower than the arrival times observed in a recent study on a production cluster [32], which reports a median arrival time of 10 us for a 1,500-server network (i.e., three times bigger than our simulated network).

[3]This assumes uniform traffic and minimal routing. With non-minimal routing, the broadcast overhead is lower.

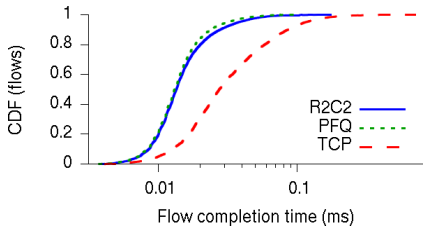


Figure 10: The CDF of FCT for short flows (size < 100 KB) for flow inter-arrival time τ=1us (log scale).

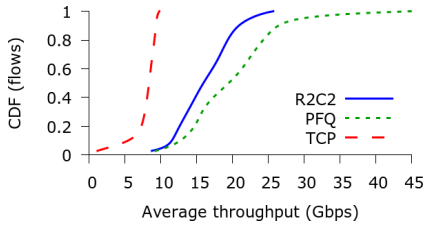


Figure 11: The CDF of average throughput for long flows (size > 1 MB) for flow inter-arrival time τ=1us.

Flow completion time and queuing. We start our analysis by measuring the flow completion time achieved by R2C2. We compare our approach against TCP and against an idealized baseline, *per-flow queues (PFQ)*, that uses backpressure and per-flow queues at each node. This baseline is impractical because, apart from forwarding complexity at rack nodes, it results in very high buffering requirements. However, it is useful in our study because it provides the upper bound of the performance achievable by any rate control protocol for minimal and non-minimal routing. For TCP, we use an ECMP-like routing protocol, which selects a single path between source and destination, based on the hash of the flow ID. This ensures that packets belonging to the same flow are routed onto the same path as required by TCP. However, we assign different shortest paths to different flows be-

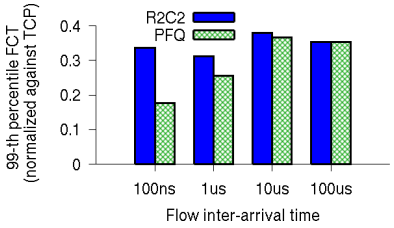


Figure 12: The 99^{th}-percentile of FCT for short flows (size < 100 KB) normalized against TCP for different flow inter-arrival times.

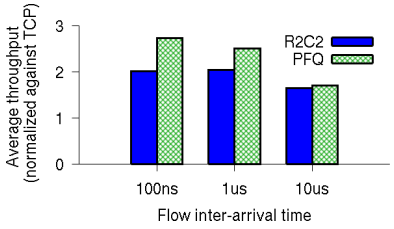


Figure 13: The average throughput for long flows (size > 1 MB) normalized against TCP for different flow inter-arrival times.

tween the same endpoints. For the idealized baseline and R2C2, we use random packet spraying.

Figure 10 and 11 show the CDF of the flow completion time (FCT) for short flows (size < 100 KB) and average throughput for long flows (size > 1 MB) respectively. We do not show the results for intermediate sizes as they are qualitatively similar to the ones for the short flows. As expected, TCP achieves the worst performance both for short and long flows. At the $99th$ percentile, TCP yields a 3.21x higher FCT for short flows and a 2.55x lower average throughput for long flows compared to R2C2. The reason is due to its high queuing occupancy (for short flows) and its dependency on a single-path routing (for long flows). The latter aspect underlines the importance of exploiting path diversity.

The results also show that for short flows R2C2 closely matches the FCT of the PFQ configuration and it does so by only using a single queue per port. This confirms that our protocol is able to achieve fairness among flows without requiring per-flow state at the intermediate nodes. This is particularly important for many data center applications, which are sensitive to load imbalance and long tail latency [21]. For long flows, the gap between R2C2 and PFQ increases. The is due to the fact that a) R2C2 uses a different fairness model that trades off utilization for computation tractability (§3.3.1) and b) it uses headroom to absorb bursts (§3.3.2).

To investigate the performance of R2C2 at different loads, in Figure 12 and 13 we show the 99^{th} percentile of the FCT (short flows) and the average throughput (long flows) normalized against TCP for different inter-arrival times τ. As expected, at very high load (τ=100 ns), the performance of R2C2 deviates from the PFQ's ideal one as the queues start building up due to the inaccuracy introduced by our periodic recomputation. However, as noted, this scenario represents an extreme case; we believe that flow inter-arrival times of the order of (tens of) micro-seconds are more realistic. As the load decreases, R2C2's performance converges to PFQ's.

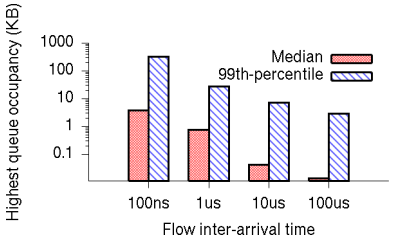


Figure 14: 99^{th} percentile and median of the maximum queue occupancy (log scale).

Figure 14 shows the median and the 99^{th} percentile of the *maximum* queue occupancy observed throughout these experiments across all node queues. For inter-arrival times of 1 us and above, the 99^{th} percentile is lower than 27 KB (median is less than one packet). However, for τ=100 ns, the 99^{th} percentile goes to 330.6 KB (median is 3.8 KB). More frequent recomputation would mitigate this, albeit at the cost of higher recomputation overhead.

Low queue occupancy also reduces the amount of packet reordering caused by multi-path routing. In this experiment, for τ=1us, the 95^{th} percentile of the re-order buffer size was 30 packets with a maximum value of 51 packets.

Impact of the recomputation interval. To understand the impact of the recomputation interval ρ, we measure the *absolute* difference between the average rate assigned to each flow when using different values of ρ and the ideal case of ρ=0. We plot the median and 95^{th}-percentile across *all* flows in Figure 15 using a flow inter-arrival time of τ=1 us (values are normalized against the ideal rates for ρ=0). As expected, lower values of ρ reduce the difference with the ideal rates. Unfortunately, such low values are hard to achieve using a software-only implementation (§5.1). The results in Figure 15 indicate that a value of ρ in the range 500 us-1 ms might be a good compromise between performance and computation overhead. Such a value incurs a modest processing overhead (see Figure 8) and contains the difference w.r.t. ideal rates within 8.2% in the median case (resp. 37.9% at the 95^{th}-percentile). This, however, depends on the workload considered, as shown in Figure 16. At lower load (e.g., τ=100 us), the difference is almost negligible, while at higher load (τ=100 ns) the difference becomes significant and lower recomputation intervals are needed. Hardware offload of the rate computation might be a viable option to enable smaller recomputation intervals but we leave the exploration of this opportunity to future work.

Impact of the headroom. To compensate for the burstiness of the workload and the inaccuracies due to the periodic rate computation, we reserve some headroom when computing the flow rates. The optimal value of headroom depends on the workload. High headroom provides more resilience to bursts but penalizes utilization. While we have not yet designed a mechanism to automatically tune it, in Figure 17 we conduct a sensitivity analysis varying the headroom from zero (i.e., no headroom) up to 20%. The figure shows that

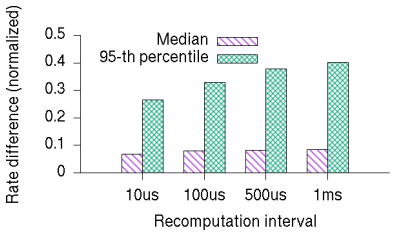


Figure 15: The median and 95th-percentile of the normalized difference between the ideal and computed rates against the recomputation interval (flow inter-arrival time τ=1 us).

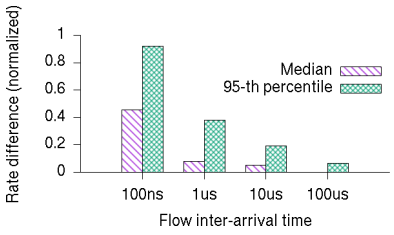
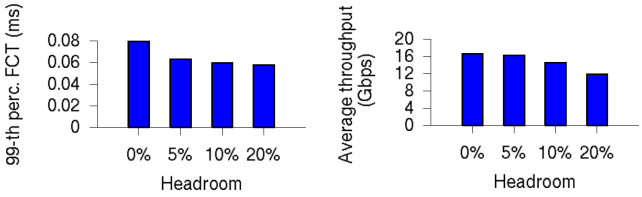


Figure 16: The median and 95th-percentile of the difference between the ideal and computed rates against the flow inter-arrival time (recomputation interval ρ=500 us).



(a) Short flows. (b) Long flows.

Figure 17: The 99th-percentile of FCT (short flows) and average throughput (long flows) against the headroom.

the performance of R2C2 is not particularly sensitive to the choice of the headroom parameter. Overall, we found that, for our target workloads, a 5% headroom represents a good trade-off. For example, for an inter-arrival time of 1 us, compared to the case in which no headroom is used, a 5% headroom yields a 21.9% reduction in FCT for short flows at the 99th percentile while the reduction in the average throughput for long flows is less than 3%.

Custom routing protocols. Next, we explore the behavior of R2C2's flexible routing stack. In particular, we aim to: i) demonstrate the benefits of having multiple routing protocols running concurrently (as opposed to a single, network-wide, routing) and ii) to evaluate the performance of our adaptive selection of routing protocols. We consider a workload in which a fraction L of nodes generates a long-running flow each to another randomly chosen node such that every node is the source and the destination of at most one flow. We chose this workload for its relative simplicity, which allows for an intuitive analysis. At high load ($L \geq$0.5), a minimal routing protocol such as RPS achieves the best performance as the hop count is minimized and the utilization is maximized. At low load, instead, a non-minimal routing protocol such as VLB exhibits superior performance as it can

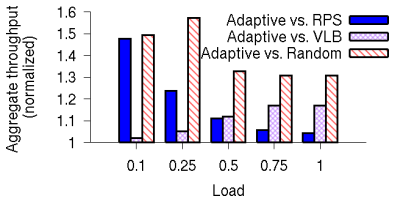


Figure 18: Aggregate throughput achieved by our routing selection heuristic (*Adaptive*) normalized against the three baselines for different load values.

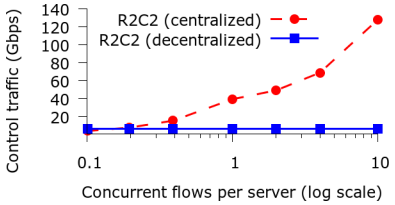


Figure 19: Control traffic with varying number of concurrent long flows per server (flow inter-arrival time τ=1 us).

exploit the spare network capacity to increase the throughput. We also experimented with other workloads and observed qualitatively similar results.

We use the genetic algorithm-based heuristic described in Section 3.4. For each flow, we consider only two routing protocols, random packet spraying (RPS) and VLB. Each genotype is encoded as a bit string with each bit corresponding to one flow, thus leading to a global search space of up to 2^{512} solutions when $L = 1$. We use a population size of 100 and a mutation probability of 0.01. In Figure 18 we plot the relative performance of our selection heuristic named *Adaptive* against three baselines: one using RPS for all flows (*RPS*), one using VLB for all flows (*VLB*), and one in which each flow randomly chooses either protocol (*Random*). The results show that our selection process is able to *always* achieve the best performance across all load values (the relative performance is always above one). This shows the importance of supporting per-flow routing protocols and the benefits of our dynamic selection.

Comparison against a centralized design. With R2C2, nodes can locally compute a flow's rate and routing protocol. We also considered an alternate design where the computation is done centrally (similar to Fastpass [36]), simply by choosing one of the rack nodes as a centralized controller. Such a design reduces computation overhead at the expense of greater control traffic. We study this trade-off below.

Figure 19 shows the amount of control traffic with a decentralized design and a centralized one when varying the number of concurrent long flows per server. In the decentralized design, a flow arrival (or departure) event is broadcasted to all rack nodes. Instead, with a centralized design, the source sends a unicast message to the controller, which computes the rates and sends to each rack node sourcing a flow a different rate message with all the new rates for its own flows. This is the reason why the control traffic for the centralized design increases with the number of concurrent flows while it remains constant for the decentralized design.

When there are only a few long flows, the centralized design is more efficient because only a few control messages are generated. However, when the number of flows grows, the decentralized becomes more attractive because only the flow events are broadcasted while the rate updates are computed locally. For example, when there is one concurrent flow per server, the centralized design generates 6.2x more traffic than the decentralized one (resp. 19.9x more when the number of concurrent flows per server is equal to 10).

The price paid by the decentralized design is increased computation. Instead of computing rates once at the controller and communicating them to all nodes, the computation is done by each node. However, as shown in Section 5.1, the rate computation overhead is acceptable.

6. DISCUSSION AND FUTURE WORK

R2C2 targets intra-rack communication, and determines how such traffic is routed and how the network fabric is shared. Here we discuss some directions for future work.

Inter-rack networking. A key open question we have not addressed is interconnection of multiple rack-scale computers to form a large cluster. This includes both the physical wiring layout and the network protocols used to bridge between two rack-scale computers.

One simple option for inter-rack networking is to just use traditional switches and tunnel R2C2 packets by encapsulating them inside Ethernet frames. While this would allow for a smooth transition from today's deployments, it has some limitations. First, given the high bandwidth available within a rack, the only way to avoid creating high oversubscription would be to use high-radix switches with large back-plane capacity, in the order of (tens of) Terabits. This, however, would dramatically increase costs and it may even be infeasible if 100+ Gbps links are to be deployed within a rack. Further, the need to bridge between R2C2 and Ethernet would increase the overhead and the end-to-end latency. A more promising (albeit challenging) solution instead is to directly connect multiple rack-scale computers without using any switch, similar to [49]. Theia [47] also proposes such design with multiple parallel connections between racks. Beside saving the cost of the switches, this would also enable a finer-grain control over the inter-rack routing.

Reliability. Even within the context of intra-rack communication, more work is needed. R2C2 does not provide a *complete network transport protocol* — it does not provide end-to-end reliability and flow control. While traditional mechanisms like end-to-end acknowledgements and checksums can be used to achieve these, we believe R2C2's design improves the efficacy of such mechanisms. For example, by decoupling congestion control from reliability, we ensure that acknowledgements are used solely for reliability. This is in contrast with TCP-like protocols that rely on ACK-clocking to determine the fair sending rates of flows. We are currently investigating such extensions.

R2C2 atop switched networks. R2C2's design is motivated by the challenges and opportunities posed by rack-scale computers with direct-connect topologies. However, traditional switched topologies (with silicon photonics or other physical technologies) are also being considered for the intra-rack network [7, 53]. We note that it is the scale of rack-scale computers, not the topology, that makes broadcasting efficient. For example, consider a 512 node rack connected using 32-port switches arranged in a two-level folded Clos topology. A broadcast on this topology results in only 8.7 KB of total traffic. Such a topology does not have multiple paths between nodes, so there is no room for route selection. However, R2C2's congestion control still offers more flexibility over traditional distributed congestion control.

At data center scale, the broadcast overhead is high and distributed control is more appealing. However, even at such a scale, R2C2's design and algorithms could be appropriate if next-generation networks have more efficient means of achieving (approximate) global visibility while providing significant multi-pathing.

7. CONCLUSION

We presented R2C2, a network stack for rack-scale computers comprising a rate-based congestion control protocol and a flexible routing mechanism. By broadcasting flow events, we ensure that rack nodes can locally compute rate allocations and routing decisions. These decisions are enforced at the sources, resulting in simplified packet forwarding. By deploying R2C2 on an emulated rack and a (cross-validated) simulator, we show that R2C2 can achieve good performance across diverse network workloads, and routing flexibility can provide even more gains.

Acknowledgments

We thank Thomas Bigger, Aleksandar Dragojević, Sergey Grant, Sergey Legtchenko, Dushyanth Narayanan, Greg O'Shea, Bozidar Radunović, Michael Schapira, the anonymous SIGCOMM reviewers, and our shepherd George Porter for their feedback and help.

8. REFERENCES

[1] H. Abu-Libdeh, P. Costa, A. Rowstron, G. O'Shea, and A. Donnelly. Symbiotic Routing in Future Data Centers. In *SIGCOMM*, 2010.

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

[3] M. Alizadeh, A. Kabbani, T. Edsall, B. Prabhakar, A. Vahdat, and M. Yasuda. Less Is More: Trading a Little Bandwidth for Ultra-Low Latency in the Data Center. In *NSDI*, 2012.

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

[5] J. M. andJeff Shamma. Revisiting log-linear learning: Asynchrony, completeness and payoff-based implementation. *Games and Economic Behavior*, 2012.

[6] S. Angel, H. Ballani, T. Karagiannis, G. O'Shea, and E. Thereska. End-to-end Performance Isolation through Virtual Datacenters . In *OSDI*, 2014.

[7] K. Asanovic. FireBox: A Hardware Building Block for 2020 Warehouse-Scale Computers. In *FAST*, 2014. Keynote.

[8] B. Awerbuch, R. Khandekar, and S. Rao. Distributed Algorithms for Multicommodity Flow Problems via

Approximate Steepest Descent Framework. *ACM Trans. Algorithms*, 9(1), 2012.

[9] S. Balakrishnan, R. Black, A. Donnelly, P. England, A. Glass, D. Harper, S. Legtchenko, A. Ogus, E. Peterson, and A. Rowstron. Pelican: A Building Block for Exascale Cold Data Storage. In *OSDI*, 2014.

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

[11] H. Ballani, K. Jang, T. Karagiannis, C. Kim, D. Gunawardena, and G. O'Shea. Chatty Tenants and the Cloud Network Sharing Problem. In *NSDI*, 2013.

[12] D. Bertsekas and R. Gallager. *Data Networks*. Prentice Hall, 1987.

[13] D. Bertsimas and J. Tsitsiklis. Simulated Annealing. *Statistical Science*, 8(1), 1993.

[14] R. S. Cahn. *Wide Area Network Design: Concepts and Tools for Optimization*. Morgan Kaufmann, 1998.

[15] M. Chowdhury and I. Stoica. Coflow: A Networking Abstraction for Cluster Applications. In *HotNets*, 2012.

[16] P. Costa, H. Ballani, and D. Narayanan. Rethinking the Network Stack for Rack-scale Computers. In *HotCloud*, 2014.

[17] Cray Inc. Modifying Your Application to Avoid Aries Network Congestion, 2013.

[18] Cray Inc. Network Resiliency for Cray XC30 Systems, 2013.

[19] A. Daglis, S. Novaković, E. Bugnion, B. Falsafi, and B. Grot. Manycore Network Interfaces for In-memory Rack-scale Computing. In *ISCA*, 2015.

[20] W. Dally and B. Towles. *Principles and Practices of Interconnection Networks*. Morgan Kaufmann, 2003.

[21] J. Dean and L. A. Barroso. The Tail at Scale. *Communications of ACM*, 2013.

[22] A. A. Dixit, P. Prakash, Y. C. Hu, and R. R. Kompella. On the Impact of Packet Spraying in Data Center Networks. In *INFOCOM*, 2013.

[23] F. R. Dogar, T. Karagiannis, H. Ballani, and A. Rowstron. Decentralized Task-aware Scheduling for Data Center Networks. In *SIGCOMM*, 2014.

[24] A. Dragojević, D. Narayanan, O. Hodson, and M. Castro. FaRM: Fast Remote Memory. In *NSDI*, 2014.

[25] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. VL2: A Scalable and Flexible Data Center Network. In *SIGCOMM*, 2009.

[26] S. Han, N. Egi, A. Panda, S. Ratnasamy, G. Shi, and S. Shenker. Network Support for Resource Disaggregation in Next-generation Datacenters. In *HotNets*, 2013.

[27] J. H. Holland. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, 1975.

[28] C.-Y. Hong, M. Caesar, and P. B. Godfrey. Finishing flows quickly with preemptive scheduling. In *SIGCOMM*, 2012.

[29] K. Jang, J. Sherry, H. Ballani, and T. Moncaster. Silo: Predictable Message Latency in the Cloud. In *SIGCOMM*, 2015.

[30] V. Jeyakumar, M. Alizadeh, D. Mazières, B. Prabhakar, C. Kim, and A. Greenberg. EyeQ: Practical Network Performance Isolation at the Edge. In *NSDI*, 2013.

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

[32] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, and R. Chaiken. The nature of data center traffic: measurements & analysis. In *IMC*, 2009.

[33] D. Nace, N.-L. Doan, E. Gourdin, and B. Liau. Computing Optimal Max-min Fair Resource Allocation for Elastic Flows. *IEEE/ACM Trans. Netw.*, 14(6), 2006.

[34] S. Novakovic, A. Daglis, E. Bugnion, B. Falsafi, and B. Grot. Scale-out NUMA. In *ASPLOS*, 2014.

[35] G. P. Nychis, C. Fallin, T. Moscibroda, O. Mutlu, and S. Seshan. On-chip Networks from a Networking Perspective: Congestion and Scalability in Many-core Interconnects. In *SIGCOMM*, 2012.

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

[37] L. Popa, G. Kumar, M. Chowdhury, A. Krishnamurthy, S. Ratnasamy, and I. Stoica. FairCloud: Sharing the Network in Cloud Computing. In *SIGCOMM*, 2012.

[38] A. Putnam, A. Caulfield, E. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J.-Y. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Y. Xiao, and D. Burger. A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services. In *ISCA*, 2014.

[39] S. Radhakrishnan, Y. Geng, V. Jeyakumar, A. Kabbani, G. Porter, and A. Vahdat. SENIC: Scalable NIC for End-Host Rate Limiting. In *NSDI*, 2014.

[40] B. Radunović and J.-Y. L. Boudec. A Unified Framework for Max-min and Min-max Fairness with Applications. *IEEE/ACM Trans. Netw.*, 15(5), 2007.

[41] C. Raiciu, S. Barre, C. Pluntke, A. Greenhalgh, D. Wischik, and M. Handley. Improving Datacenter Performance and Robustness with Multipath TCP. In *SIGCOMM*, 2011.

[42] T. Roughgarden and E. Tardos. How Bad is Selfish Routing? *J. ACM*, 2002.

[43] B. Schroeder and G. A. Gibson. Understanding Failures in Petascale Computers. *Journal of Physics*, 78, 2007.

[44] A. Singh, W. J. Dally, B. Towles, and A. K. Gupta. Locality-preserving Randomized Oblivious Routing on Torus Networks. In *SPAA*, 2002.

[45] L. G. Valiant and G. J. Brebner. Universal Schemes for Parallel Communication. In *STOC*, 1981.

[46] B. Vamanan, J. Hasan, and T. N. Vijaykumar. Deadline-Aware Datacenter TCP (D^2TCP). In *SIGCOMM*, 2012.

[47] M. Walraed-Sullivan, J. Padhye, and D. A. Maltz. Theia: Simple and Cheap Networking for Ultra-Dense Data Centers. In *HotNets*, 2014.

[48] C. Wilson, H. Ballani, T. Karagiannis, and A. Rowtron. Better Never Than Late: Meeting Deadlines in Datacenter Networks. In *SIGCOMM*, 2011.

[49] H. Wu, G. Lu, D. Li, C. Guo, and Y. Zhang. MDCube: A High Performance Network Structure for Modular Data Center Interconnection. In *CoNEXT*, 2009.

[50] Amazon joins other web giants trying to design its own chips. http://bit.ly/1J5t0fE.

[51] Boston Viridis Data Sheet. http://bit.ly/1fBnsQ9.

[52] Calxeda EnergyCore ECX-1000. http://bit.ly/1nCgdHO.

[53] Design Guide for Photonic Architecture. http://bit.ly/NYpT1h.

[54] Google Ramps Up Chip Design. http://ubm.io/1iQooNe.

[55] How Microsoft Designs its Cloud-Scale Servers. http://bit.ly/1HKCy27.

[56] HP Moonshot System. http://bit.ly/1mZD4yJ.

[57] Intel Atom Processor D510. http://intel.ly/1wJmS3D.

[58] Intel, Facebook Collaborate on Future Data Center Rack Technologies. http://intel.ly/MRpOM0.

[59] Intel Rack Scale Architecture. http://ubm.io/1iejjx5.

[60] Maze: A Rack-scale Computer Emulation Platform. http://aka.ms/maze.

[61] RDMA Aware Networks Programming User Manual. http://bit.ly/1ysVa1O.

[62] SeaMicro SM15000 Fabric Compute Systems. http://bit.ly/1hQepIh.