# Hierarchical Congestion Control (HCC): Fairness and Fast Convergence for Data Centers

Shiva Ketabi, Yashar Ganjali
*Department of Computer Science, University of Toronto*

***Abstract*—Congestion control protocols face several challenges for achieving max-min fairness and high throughput in a timely manner. In the absence of explicit signals, flows infer the congestion state solely based on their limited view of the network. Any attempt to expand the view with support from a centralized entity or network switches needs to deal with overheads as the number of flows scales.**

**We present a system called Hierarchical Congestion Control (HCC) for data center networks. Flows that share bottlenecks are iteratively aggregated to build a hierarchy for propagating congestion control signals. HCC enables flow cooperation for sharing congestion information and collectively reacting to congestion. Aggregate control of multiple flows improves fairness and helps flows to converge faster in case of fluctuations in flow demands or network state. To limit the overheads, we compress congestion signals as they move up the hierarchy, pruning signals that do not involve significant changes.**

**Our evaluation on real workload shows HCC converges to full utilization very fast with an empty queue and near-optimal fairness. Moreover, comparing to a centralized solution, HCC reduces the communication overhead to the core by $73\%$ and the communication overhead from the core by $99.5\%$.**

## I. Introduction

A common weakness of most congestion control protocols is lack of cooperation and information sharing among flows. Congestion control protocols that are distributed by design delegate the task of congestion detection to individual flows (*e.g.* [1]–[7]). These individual flows are required to estimate the congestion state based on limited congestion signals they receive. These signals, ranging from packet loss (*e.g.* TCP Reno [1]) to variations in packet round-trip latencies, (*e.g.* Swift [7] and BBR [6]), and even direct signals of congestion (*e.g.* DCTCP [5]) are intrinsically inaccurate given the limited information they carry (typically single or a few bits of information) about the congestion state.

This limited view of individual flows and lack of coordination among flows make reaching the optimal state challenging. In particular, it is extremely difficult to reach optimum fairness and throughput in short periods of time, especially for short flows. In contrast, let us imagine a congestion control framework which has the global view of the congestion experienced by all flows. For example, we can have a logically centralized controller (*e.g.* [8], [9]) in the network that collects congestion signals from all the flows and globally allocates packet injection rates to them (*e.g.* by solving an optimization problem aiming at maximizing throughput while respecting fairness). Unfortunately, such a centralized congestion controller might be infeasible due to scalability challenges when it comes to gathering and processing information from a large number of flows, and sending updates back in a timely manner.

ISBN 978-3-903176-48-5 

Aiming for the best of both worlds, we propose keeping sub-optimal but prompt reaction of individual flows, and adding aggregate control of multiple flows on top. We believe that merely relying on individual flows and their limited feedback from the network leads to sub-optimal throughput and poor fairness, especially in short periods of time. However, we can significantly alleviate this problem if we provide a mechanism for flows to share their knowledge of the congestion state with one another with low communication overhead, and derive a more broad and accurate view of the network.

To this end, we propose Hierarchical Congestion Control (HCC), a system and a protocol for hierarchical aggregation and distribution of congestion signals among flows sharing bottlenecks in data center networks. The intuition here is that flows sharing bottlenecks can share their knowledge of the congestion state of these bottlenecks. We focus on data center networks since cooperation among different agents (and flows) is easier to handle as the network is typically under a single authority. Also, optimal rate allocation and timely reaction to changes are extremely important due to stringent requirements of data center applications, making it a perfect match for HCC.

Our solution has two main phases: flow aggregation and rate allocation. In flow aggregation, a group of flows sharing bottlenecks are aggregated into a super-flow which itself can be an input to another iteration of the flow aggregation. The result of the flow aggregation, after a number of iterations, is a hierarchy of flows and super-flows used for rate allocation (Figure 2 and Section III-B). Given the hierarchy of flows and super-flows, we assign a rate allocation agent to each flow/super-flow, with two responsibilities. First, each agent receives updates about the allocated share from its parents in the hierarchy and accordingly calculates the effects on the fair share of its children. Second, it gathers updates from its children about congestion signals or demands and propagates the aggregated changes to its parents.

A core design principle in HCC is that as congestion signals are elevated through the hierarchy, they are compressed to ensure that the communication overhead and the state size are kept limited at the core of the network. In the other direction, the signals will be disaggregated as moving down the hierarchy to provide expanded signals and more precise information to individual flows. To limit the communication overhead and delay, we leverage the hierarchical design to aggregate, prune, and quantize the messages as they move toward the core of the network. We will explain the details in Section III-C.

What follow are the main contributions of HCC:

**Aggregate control of multiple flows.** The proposed hierarchy of flows enables cooperation among flows through exchanging congestion control signals. As a result, flows will be provided with more informative and accurate signals rather than a single or a few bits of information leading to significant improvement in fairness of the rate allocation scheme.

**Limited flow state size and communication overhead at the core.** A key design decision in HCC is to quantize the signals. Combining that with flow aggregation ensures that the size of the state does not grow with the number of flows as we go up the hierarchy. This keeps the state space of the rate allocation agents limited and reduces the communication overhead at the core agents, and is a key enabler for our work.

**Fast reaction to frequent changes at the edge.** Adding local agents on top of the individual flow congestion control leads to fast and fair reaction to fluctuations at the edge with cooperation of local agents. Also, the local agents can be very helpful for short-lived flows to provide them with better initial rates and for on-off traffic patterns to inform the flows about the fair shares when they resume.

For the evaluation, we compare HCC with well-known congestion control protocols (NewReno [10], Cubic [11], and DCTCP [5]), given real data distributions ( [5], [12]). For web search workload, we achieve up to 4x faster convergence to the full link utilization, near zero bottleneck queue size, $42-62\%$ reduction in mean flow completion time, and fairness index of more than $0.9$ in $89\%$ of the time. To evaluate the communication and computation overheads, we compare HCC with a centralized solution. HCC reduces the communication overhead to the core by $73\%$ and the communication overhead from the core by $99.5\%$. Besides the significant reduction in the overhead, HCC's local presence results in fast reaction at the edge compared to a centralized solution.

## II. Related Work

**Data center-specific solutions.** There have been a large number of transport solutions specialized for data center networks (*e.g.* [4], [5]). Benefiting from a global view of the network state, a line of work aims at achieving the ground truth for rate allocation in a centralized manner [8], [13]. However, any centralized congestion control solution needs to address the huge transmission cost and delay required for collecting and distributing information, as well as the processing time for a large number of flows in a single point of control. On the other hand, DCTCP [5], the widely-used congestion control protocol for data centers, resorts to individual flows estimating congestion state of the network through Explicit Congestion Notification (ECN) signals, but without sharing their state with one another. Moreover, successors of DCTCP (*e.g.* [14], [15]) require extra information (flow deadlines and flow sizes) and/or changes to switches. An orthogonal direction to HCC is the flow scheduling mechanisms implemented at the switches that minimize short-lived flow completion times for data centers (*e.g.* [16], [17]) which mostly require custom design at switches. HCC enables distributed max-min fairness and aggregate control of multiple flows without changing switches, knowing flow deadlines or flow sizes, relying on any specific scheduling mechanism, and imposing per-flow overheads. A closely-related prior work to HCC [18] provides a distributed arbitration for rate allocation and leverages the data center structure for reducing the overheads, but the main difference is that it does not use flow aggregation for reducing the overheads and realizing max-min fairness as HCC.

**Flow aggregation.** RackCC [19] is a simple and elegant attempt to aggregate flows for congestion control in a data center. It uses tunneling to aggregate flows between pairs of top of rack switches (*i.e.* ToR-ToR traffic). Rack-level flow aggregation helps flows to combine their limited knowledge of the congestion state and to collectively agree on their optimum rates. Moreover, new flows, with no prior knowledge of the network, can acquire the congestion state of older flows and immediately set a near-optimal sending rate instead of wasting time searching for the optimum rate. RackCC provides a single layer of aggregation at ToR switches. In contrast, HCC allows arbitrary aggregation and a hierarchical structure that enables explicit rate allocation using max-min fairness.

**In-network intervention and credit-based solutions.** A group of solutions propose explicit rate allocation using in-network intervention [20]–[22], or receiver side credits [23], both incurring significant overheads and reacting at Round-Trip Time (RTT) scales. In contrast, HCC's hierarchical design obviates the need for per-flow communication overhead and enables fast (sub-RTT) reactions. Also, HCC does not require any changes to switches: we can deploy our agents on distributed control plane instances with presence close to certain switches. Moreover, a major drawback of fair queuing schemes at switches (*e.g.* [24], [25]) is they require switches to store a state space relative to the number of flows. A recent interesting work [26] guarantees max-min fairness among tenants of a multi-tenant cloud without storing the state of individual flows at switches (still requires changes to switches). However, providing max-min fairness at the flow level (not tenant level) in a global manner remains an open problem there.

## III. System design

In this section, we first introduce our model for computing the max-min fair rate allocation given a set of flows and bottlenecks in a centralized manner (Section III-A). We then expand this model to a distributed architecture to make it practical by removing the need for a single point of control and reducing the communication and computation overheads. We explain how we build our hierarchical architecture of flows in the flow aggregation phase in Section III-B, and elaborate on how this hierarchical architecture is used for rate allocation in a distributed and efficient manner in Section III-C. Finally, we mention the implementation details in Section III-D.

### A. Model

Let us consider a data center topology in which the set of all flows in a given time is represented by $\mathcal{F} = \{f_1, f_2, \ldots, f_N\}$. For each flow $f_j \in \mathcal{F}$, we represent its source, destination, and a path between the source and destination by $(s_j, d_j, p_j)$. We assume there is a limited number of bottleneck links in the network $\mathcal{B} = \{b_1, b_2, \ldots, b_M\}$.

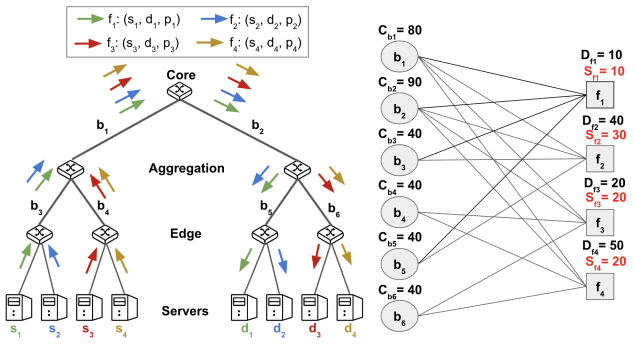



Fig. 1. An example of a three-tier topology, and its bottleneck-flow graph.

**Bottleneck-flow graph.** To model the inter-dependence between flow rates and bottlenecks, we consider a bipartite graph with vertices on the left representing bottlenecks in the network, and vertices on the right representing flows. Each vertex on the left has an associated capacity and each vertex on the right has an associated demand and share. We connect any vertex $f_j$ on the right to any vertex $b_i$ on the left if flow $f_j$ passes through the bottleneck $b_i$, *i.e.*, $b_i \in p_j$. For a bottleneck vertex $b_i$, $\mathcal{N}(b_i)$ denotes the set of neighbours of $b_i$ in the bipartite graph, *i.e.*, the set of flows sharing bottleneck $b_i$.

A given rate allocation at time $t$ is represented by assigning weights to edges. An edge with weight $w_{i,j}$ between a bottleneck $b_i$ on the left and a flow $f_j$ on the right shows that flow $f_j$ has received $w_{i,j}$ units of capacity from bottleneck $b_i$. Also, the capacity of a bottleneck $b_i$ is denoted by $C_{b_i}$ and the demand of a flow $f_j$ is denoted by $D_{f_j}$. The goal is to find the shares of the flows denoted by $S_{f_j}$. Figure 1 shows an example of a three-tier network topology and its associated bottleneck-flow graph.

**Single bottleneck max-min fairness.** Let us show how we can use the bottleneck-flow graph to calculate the max-min fairness allocation. We start with the straightforward case in which the goal is to find the max-min fair allocation for a single bottleneck ($\mathcal{B} = \{b_1\}$). Algorithm 1 illustrates a simple solution for the single bottleneck case in the bottleneck-flow graph representation.

**Multiple bottleneck max-min fairness.** The above algorithm allocates the capacity of a bottleneck to its associated flows in a max-min fair manner. We use this algorithm to calculate the max-min fair rate allocation in the general case with multiple bottlenecks. Algorithm 2 presents the max-min fairness algorithm for multiple bottlenecks using the bottleneck-flow graph. Initially, we start with all the flows as unallocated, *i.e.* their fair shares are not yet determined. At each iteration, we distribute the capacity of bottlenecks which still have available capacity to their neighbours using Algorithm 1. Then, we find the edge with the lowest weight corresponding to the lowest allocation and select its connected flow. The rate for this selected flow can be finalized at this iteration. Therefore, we set the fair share for this edge and return any residual capacity to the associated bottlenecks. Finalizing the max-min fair share of each flow at each iteration guarantees the termination of the algorithm in a max-min fair state.

**Incremental max-min fairness algorithm.** We can also extend the above algorithm to an incremental algorithm, *i.e.*, computing fair rates once and adjusting them when changes in demands or flows happen. Let us assume the network is in a stable max-min fairness state at time $t$, and then the demand of flow $f_j$ is increased by $r$ units. To see if this update leads to any changes in the rate allocated to $f_j$, we need to consider all edges connecting $f_j$ to bottlenecks in the left side of the bipartite graph. For each of these edges, we should compare its weight with weights on other edges sharing the same bottleneck on the left. If in any of these comparisons the weight appears as the maximum number, the rate increase will not be allowed since it is against the max-min fairness policy.

**Algorithm 1:** Single Bottleneck Max-Min Fairness

**Data:** $G$, $b_i$ /* The bottleneck-flow graph, and a specific bottleneck */
**Result:** $(w_{i,j}), \forall j \in \mathcal{N}(b_i)$ /* Rate allocations */
/* Sort demands in increasing order */
$(x_1, x_2, \dots x_X)$ = Sort($(D_{f_j} | f_j \in \mathcal{N}(b_i))$);
**Init:** $y = 1$;
/* Find fair rate of flows one by one */
**while** $y \leq X$ **do**
  /* If a flow asks for less than its fair share, allocate all of its demand; otherwise, allocate the fair share */
  **if** $x_y < \frac{C_{b_i}}{X-y+1}$ **then**
    $w_{i,y} = x_y$;
  **else**
    $w_{i,y} = \frac{C_{b_i}}{X-y+1}$ ;
  **end**
  $C_{b_i} - = w_{i,y}$;
  $y + = 1$ ;
**end**

**Algorithm 2:** Multiple Bottleneck Max-Min Fairness

**Data:** $G$ /* The bottleneck-flow graph */
**Result:** $S_{f_j}, \forall\ 1 \leq j \leq N$ /* Rate allocations */
**Init:** $\mathcal{F}' = \mathcal{F}$;/* Set of unallocated flows */
/* Find fair rate of flows one by one */
**while** $\mathcal{F}' \neq \emptyset$ **do**
  /* Run Algorithm 1 on bottlenecks with available capacity */
  **foreach** $b_i \in \{b_i | 1 \leq i \leq M, C_{b_i} \neq 0\}$ **do**
    $(w_{i,j})$ =SingleBottleneckMaxMinFairness$(G, b_i)$;
  **end**
  /* Select $f^*$ with lowest allocation */
  $w^* = \min_{1 \leq i \leq M, f_j \in \mathcal{N}(b_i), f_j \in \mathcal{F}'} w_{i,j}$;
  $i^*, j^* = \arg\min_{1 \leq i \leq M, f_j \in \mathcal{N}(b_i), f_j \in \mathcal{F}'} w_{i,j}$;
  $f^* = f_{j^*}$;
  /* Return residual capacity of selected flow to associated bottlenecks */
  **foreach** $b_i \in \{b_i | f_{j^*} \in \mathcal{N}(b_i)\}$ **do**
    $residual_i = w_{i,j^*} - w^*$;
    $C_{b_i} + = residual_i$;
  **end**
  /* Set share of selected flow and remove it from unallocated set */
  $S_{f^*} = w^*$;
  $\mathcal{F}' = \mathcal{F}' - f^*$;
**end**

Otherwise, depending on the minimum increase allowed by all the bottlenecks, a fraction of $r$ will be added to the share of flow $f_j$. Also, other flows sharing the same bottlenecks with $f_j$ that have higher weights on their associated edges to the bottlenecks will lower down their rates. This decrease in the rates of the flows can recursively increase the rate of other related flows. Changes to the capacity of bottlenecks can be handled in a similar manner. Addition of new flows can be simply handled by adding a node with demand zero, and then updating the demand to the desired value. Similarly, removing a flow is handled by reducing its associated demand to zero.

We can prove that if the network is initially in a max-min fair state, this algorithm will terminate, and the resulting rate allocation will also be max-min fair. To see this, we note that each round of the algorithm can be considered as traversing paths on the bipartite graph to move a fraction of a bottleneck link's bandwidth from flows with higher shares of the bandwidth to flows with lower shares (with the condition that their demands increase). For any pair of flows $f$ and $f'$ sharing a bottleneck link, if there exists a path from $f$ to $f'$ that the algorithm traverses in the bipartite graph, the demand of $f$ must be greater than its fair share, and the demand of $f'$. Having a loop (*i.e.*, a path from $f$ to $f'$ and another from $f'$ to $f$) would be a contradiction. Upon the termination of the algorithm, we will not be able to increase any flow share without reducing the share of a flow with lower share (otherwise the algorithm would have continued), which is the definition of max-min fairness. We can show that each node is not visited more than once, and therefore, the algorithm has a run-time complexity of $O(N)$ in the worst case.

**Challenges.** The introduced algorithms can be run in a centralized entity to find the max-min fairness allocations. However, similar to any centralized solution, it lacks scalability and imposes a huge communication overhead and delay. To overcome these challenges, we build a hierarchy of flows based on their common paths, and use a distributed algorithm for rate allocation. In the following sections, we extend the definition of the bottleneck-flow graph to support a hierarchy of flows and revisit the max-min fairness algorithms.

### *B. Flow aggregation*

We note that many flows share most of the critical partitions of their paths with other flows. This observation has some useful implications for us. We can consider flows sharing a path partition as groups and consolidate the communication messages required for demand exchange and rate allocation. As a result, we can reduce the communication and computation overheads. Moreover, these groups of flows sharing their critical partitions can locally cooperate for adjusting their rates and resulting in a smoother aggregate behaviour.

**Flow aggregation model.** The above observation leads us to a model for flow aggregation. The goal is to iteratively merge flows that share common path partitions in the network and build a model to optimize the rate allocation. Let us assume the set of all flows in a timestamp, $\mathcal{F} = \mathcal{F}^0 = \{f_1^0, f_2^0, \ldots, f_N^0\}$, as input where the 0 superscripts mean that we are at iteration 0 of the aggregation phase. It generates a vertex per each $f_j^0 \in \mathcal{F}^0$ on the right side of the bottleneck-flow graph. Next, at each iteration $l$, it starts with the set of flows at the previous iteration, *i.e.* $\mathcal{F}^{l-1}$, chooses a number of flows which have shared paths, and merges them into new set of flows $\mathcal{F}^l$. At each iteration $l$, the algorithm chooses groups of flows among all the flows that share a path partition to be merged. The decision on how to choose these groups will be discussed later.

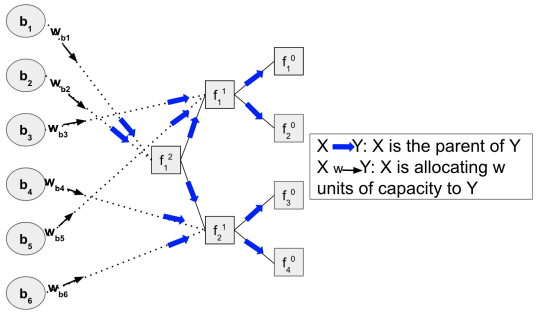



Fig. 2. An example of a flow aggregation hierarchy.

In each iteration, a new layer of vertices is added to the bottleneck-flow graph. The flows in each group $f_{j_1}^{l-1}, f_{j_2}^{l-1}, \ldots, f_{j_J}^{l-1}$, are consolidated into a single flow $f_j^l$. We create a new vertex in the bottleneck-flow graph for $f_j^l$ and connect it to the vertices $f_{j_1}^{l-1}, f_{j_2}^{l-1}, \ldots, f_{j_J}^{l-1}$ as their parent. For each flow $f_j^l$, we keep track of its source, destination, and a path between the source and destination, represented by $(s_j^l, d_j^l, p_j^l)$. For the merged flow, the associated path is the path shared amongst all its children, and the first and last node of this path are considered to be the source and destination of the flow. The aggregation phase is finished when no more flows can be aggregated, or when $l$ reaches a predefined threshold $l_{\max}$. We explain each of these termination conditions:

**Case 1.** Let us assume the aggregation is terminated at the $k$-th iteration since there are no shared links among flows $f_1^k, f_2^k, \ldots, f_{N_k}^k$. Next, we create vertices on the left of the bottleneck-flow graph for bottlenecks and connect them to the super-flows that pass through these bottlenecks. Here, each bottleneck can belong to only one of the super-flows. Therefore, we can assign the total bandwidth of the bottleneck to the corresponding super-flow, and recursively share the bandwidth amongst its children. Since each flow in level $0 \leq l < k$ might be included in more than one flow in level $l+1$ (*i.e.* might have more than one parent), it picks the minimum assigned rate among all its parents as its rate and report any excess bandwidth to its parents to be redistributed to other flows. The bottleneck-flow graph in Figure 1 is modified to Figure 2 to reflect the result of the flow aggregation algorithm.

**Case 2.** If the iterations stop with $l = l_{\max}$, we can use known congestion control algorithms to allocate flow rates to the super-flows generated by the flow aggregation phase. In other words, the set of flows in this iteration, *i.e.* $F^l = \{f_1^l, f_2^l, \ldots, f_{N_l}^l\}$, can be considered as the only flows in the network and we can run known congestion control schemes to allocate rates to them. RackCC [19] is a simple example here where $l_{max} = 1$. In this paper, we implement the first termination case, and study the second one for our future work as it goes beyond the scope of this paper.

**Flow aggregation decision.** At each iteration of flow aggregation, groups of flows with shared paths are selected and consolidated into a single flow in the higher level. At each level, we need to decide which sets of flows to select and merge among all possible sets of flows with shared path partitions. We propose the following two approaches:

- **Locality-based aggregation.** Typical data center topologies, such as trees and fat-trees [27], provide a natural way of flow aggregation. At each height of the tree, flows multiplexing together in the physical topology can be aggregated in the bottleneck-flow graph. The example in Figures 1 and 2 shows this type of aggregation. Locality-based aggregation has two main benefits: first, the flow aggregation is a straightforward procedure statically defined by the network topology; second, due to the locality-based aggregation, we can optimize the communication delay of messages.
- **Correlation-aware aggregation** Besides the network topology, the correlation among the flows can also be useful in the flow aggregation decision ( [28], [29]). Intuitively, the central limit theorem [30] suggests by aggregating an adequately large number of independent flows, we will have a smooth and predictable distribution of demand rates, as opposed to significant fluctuations in individual flows. For inversely correlated flows, the aggregate might be even smoother and thus more predictable. This intuition explains why HCC agents can properly perform without the need for frequent communications with their ancestor agents. Therefore, for a correlation-aware flow aggregation, the lower the correlation of any pair of flows, the higher their chance of being aggregated in the same group.

### C. Rate allocation

Our distributed algorithm is executed through cooperation of a set of agents namely rate allocation agents. Each instance of the agent corresponds to a vertex in the flow aggregation hierarchy. The physical placement of agents should be close to the sources of the associated flows (or super-flows) to reduce the communication delay. As defined in Section III-B, the source of each super-flow is the first node on the shared path among its children.

The rate allocation agents cooperate with one another to propagate the end-host signals, *i.e.* aggregated flow demands, from the end-hosts to the core and the capacity allocations from the core to the end-hosts in the reverse direction. The former starts with individual flows, corresponding to vertices at the right of the flow aggregation hierarchy, sending updates in their demands to their parent agents. The later is initiated by the roots of the graph, located at the left-most side of the graph, occupying the total capacity of the associated bottlenecks.

Each rate allocation agent should store the connections to its parent agents and child agents respectively denoted by Parents and Children. If the agent has no parent, *i.e.* is associated with a vertex at the left most side of the flow aggregation hierarchy, the flag IsRoot will be set for the agent. Also, if the agent has no child, *i.e.* it is associated with a vertex at the right most side of the flow aggregation hierarchy, the flag IsLeaf will be set. Each parent fairly distributes its capacity (share) among its children and each child keeps track of its Share, which is the minimum of the capacity allocated to this agent by its parents and the capacity of the links local to the agent. For a root agent, Share corresponds to the capacity of a core bottleneck, and for a leaf agent, Share corresponds to the allocated rate of an individual flow which we periodically aim to calculate and is the main objective of the entire system. Also, each agent keeps track of Demand for its descendant flows aggregated from its child agents. For leaf agents, Demand represents the associated individual flow's demand. As the demand signals are pushed to higher layers of the hierarchy, we compress them to keep the required state size and communication overhead bounded.

Algorithm 3 shows a high-level description of rate allocation for an agent. It periodically runs two parallel tasks: (i) processing signals from children (ReadLeafUpdate or ReadChildSignals), aggregating the signals in the local Demand variable (AggregateChildSignals), and sending updates to children (ReportChildren) every $\text{Cycle}_{child}$; (ii) processing signals from parents (ReadRootUpdate or ReadParentSignals), disaggregating the signals (DisaggregateParentSignals), and sending updates to parents (ReportParents) every $\text{Cycle}_{parent}$. The first task runs much more frequent, *i.e.* $\text{Cycle}_{child} << \text{Cycle}_{parent}$. Also, the frequency of running these tasks is higher for agents in lower aggregation layers. The reason is that agents at lower aggregation layers are responsible for smaller number of flows and observe more volatility in their input. Conversely, agents at higher aggregation layers, corresponding to core bottlenecks, include a large number of descendent flows but with smoother aggregate behaviour.

Depending on the termination case in Section III-B, the signals sent through the flow aggregation hierarchy can be explicit or implicit. For the first case, namely *hierarchical*

**Algorithm 3:** Rate Allocation Algorithm

```
Data: Parents, Children, IsRoot, IsLeaf, Cycle_child, Cycle_parent
Result: Share of leaves (allocated rates of individual flows)
do in parallel
    while True do
        if IsLeaf then
            ReadLeafUpdate();
        else
            ReadChildSignals();
            Demand = AggregateChildSignals();
            ReportChildren(Share);
        end
        Wait for Cycle_child;
    end
    while True do
        if IsRoot then
            ReadRootUpdate();
        else
            ReadParentSignals();
            Share = DisaggregateParentSignals();
            ReportParents(Demand);
        end
        Wait for Cycle_parent;
    end
end
```

*max-min fairness*, explicit notifications are used *i.e.* demands are sent from child agents to their parents and injection rates are sent in the reverse direction. Each rate allocation agent disaggregates its share using Algorithm 1. Similar to any explicit max-min fair rate allocation, major benefits are optimal fair allocation and fast convergence. The communication costs and delay of explicit approaches, however, are mitigated using the flow aggregation model.

For the second case, which we call *congestion control of super-flows*, implicit notifications are used *e.g.* RackCC uses ECN bits as congestion signals. Here, rate allocation agents should be able to infer the implicit congestion signals and fairly distribute the inferred signals among their children. Please note that in this paper we implement and evaluate the first case (hierarchical max-min fairness) and leave *congestion control of super-flows* as a future work.

**Overhead reduction.** A major challenge is that the input to an explicit max-min fairness algorithm is the demand rates of all individual flows at each point of time and the output is the allocated rates which means a huge number of messages should be transferred among the rate allocation agents.

We use multiple techniques to significantly reduce the size and the number of messages. First, the agents can simply aggregate messages from their children towards their parents. For instance, if an agent receives $K$ report messages from its descendant flows with the same requested demand of $D$, it can merge them into a single report message to its parents informing them of $K$ individual flows with demand $D$ instead of sending $K$ separate messages. This simple technique becomes even more effective using our next solution for message compression, which is rate quantization.

As suggested in [31], [32], rate quantization, an effective technique to group flows whose values are close, significantly reduces the run-time of max-min fairness algorithms. In HCC, we use the same rate quantization technique to reduce the computing time of max-min fairness in agents, as well as to compress the report messages traversing the hierarchy. Agents store the quantized flow demands of their descendants locally in Demand variable and send updates in quantized flow demands to their parents. Also, agents only need to send the total share of each child (a single value) since the child can distribute the share having the locally-stored Demand.

Finally, we use a pruning technique to eliminate reporting negligible updates. In other words, if an agent receives updates in demands or allocated shares whose values of changes are less than a threshold defined for each agent, we simply prune those messages. This is particularly effective when we have grouped inversely-correlated flows together. As a result, aggregation, quantization, and pruning techniques help us to keep the size of the reported messages limited.

### D. Implementation details

We have implemented HCC in C++ and ns3 simulation environment [33]. Rate allocation agents (non-leaf agents) are located close to the associated switches on the control path (no changes to the datapath). The control plane provides the root rate allocation agents with the capacity of the connected bottleneck(s) in the flow aggregation hierarchy. Agents associated with the leaves of the flow aggregation hierarchy are placed at the end-hosts and interact with the congestion control protocol at the end-hosts. They sit between the transport and application layers, receive the application layer demands as input, and adjust the rate of the congestion control protocol. For other transport functionalities such as reliability and out-of-order packet detection, we use TCP mechanisms and HCC only intervenes in rate adjustment.

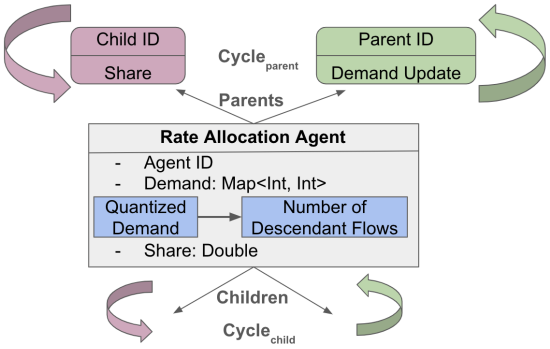



Fig. 3. Rate allocation implementation.

Figure 3 presents the high-level structure of each implemented rate allocation agent. Each agent locally stores the quantized flow demands of its descendants in a map (Demand) of demand values to the number of flows with these demand values. Also, it stores a single value, *i.e.* Share, for its total allocation. The green box shows the structure of messages from children to parents including the parent's ID and a map containing the quantized demand updates. Please note that agents only send messages if there are any updates. For example, if a flow demand changes from $D_1$ to $D_2$, the entry $\{D_1 : -1, D_2 : +1\}$ will be added to the demand update map of the message to the parents, meaning that a flow has been removed from quantized value $D_1$ and a flow has been added to quantized value $D_2$. The purple box presents the structure of the messages to the children which simply includes the child ID and a single value for the share of the child.

## IV. Evaluation

We evaluate HCC in comparison with well-known congestion control protocols, TCP NewReno [10], Cubic [11], and DCTCP [5]. We report performance metrics including throughput, flow completion time, bottleneck queue size, fairness, and packet latency along with measures for HCC's inevitable communication and computation overheads. We also compare HCC's overheads with a centralized explicit rate allocation solution. Our experiments are performed in ns3 network simulation environment [33]. We use a three-tier topology for the simulations that is a scaled version of Figure 1 with 1 core switch, 2 aggregation switches, 8 edge switches, and 64 servers. The end-hosts at the left of the three-tier topology are flow sources and the end-hosts on the right are flow destinations. HCC agents are present at aggregation switches, edge switches, and end-hosts, and their child-parent relations follow the structure of the topology similar to Figure 2.

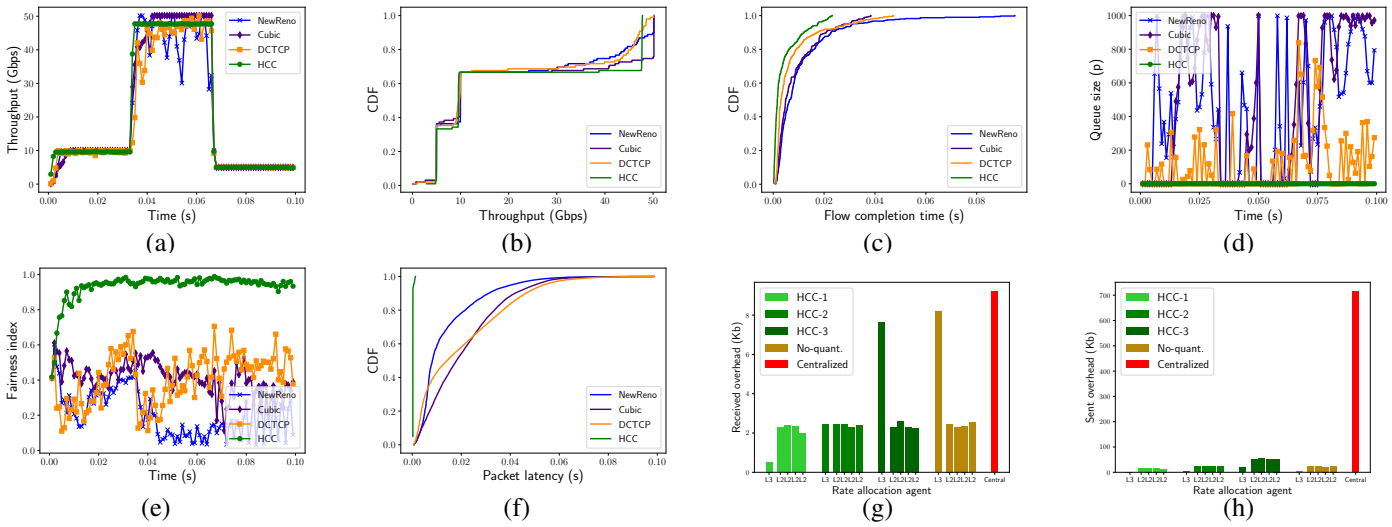



Fig. 4. Figures (a)-(f): Comparison of HCC with TCP variants in terms of throughout, flow completion time, queue size, fairness, and packet latency for web search workload. Figures (g)-(h): Comparison of HCC with a centralized rate allocation solution in terms of communication overhead and effects of quantization and value of Cycle for rate allocation agents in HCC.

The experiments are divided into two scenarios based on different workloads. In the first set of experiments, we use web search workload from production data centers [5]. This workload consists of flows generated from a distributed range of flow sizes mostly short-lived flows. The second set of experiments are based on remote procedure call (gRPC) traffic patterns observed in a production network [12] and include flows with an on-off and bursty behaviour.

### A. Web search workload

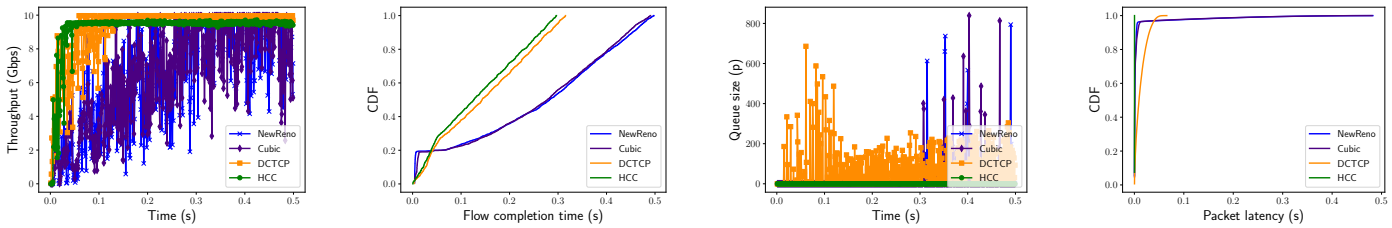
We use web search workload from production data centers [5], which is commonly used in similar evaluations. The Poisson flow injection rate is chosen based on the capacity of the network such that flows congest the network with average loads of $50\%, 10\%, 100\%$ during the simulation. The bottleneck bandwidth $B_{core}$ (core-aggregation link) is initially set to $10Gbps$ and is changed to $50Gbps$ and $5Gbps$ (at $\frac{1}{3}$ and $\frac{2}{3}$ of simulation time respectively) to examine how the protocols react to changes in the background traffic and the available bandwidth in a data center environment. Other link bandwidths are set to $B_{agg} = B_{edge} = 50Gbps$ ($B_{agg}$ is aggregation-edge bandwidth and $B_{edge}$ is server-edge bandwidth), and the link latencies are set to $L_{core} = 100$, $L_{agg} = 10$, and $L_{edge} = 1$ microseconds ($L_{core}$ is core-aggregation latency, $L_{agg}$ is aggregation-edge latency, and $L_{edge}$ is edge-server latency). The buffer size is 1000 packets, relative to the bandwidth-delay product.The total simulation time is 0.1 second. Longer simulations did not change the observed results. Initial congestion window is set to 10.

**Throughput.** Figure 4(a) shows the total throughput of all flows captured every 1 millisecond. As shown, HCC acquires almost all the capacity all the time while other protocols are slow in filling the bandwidth specially when the available bandwidth is increased to $50Gbps$. The main reason is that HCC, as an explicit max-min fairness solution, can update flows about changes in sub-RTT time scales while other protocols require several RTTs. Another observation here is that the total throughput for HCC has less variations compared to NewReno, Cubic, and DCTCP. To better understand the throughput gap, we plot the cumulative distribution function (CDF) of total throughput over time in Figure 4(b). The bottleneck bandwidth in $\frac{1}{3}$ of the time is set to $5Gbps$, in $\frac{1}{3}$ of the time is set to $10Gbps$, and in $\frac{1}{3}$ of the time is set to $50Gbps$. In all three regions, HCC converges to the full utilization faster than others (up to 4x faster). Noticeably, in the $50Gbps$ region, the utilization gap is up to $20\%$ of the capacity. Cubic performs better than DCTCP and NewReno in terms of total throughput, but as we will discuss, it is at the cost of higher queue sizes and packet latencies.

**Flow completion time.** A metric of interest for this dataset is flow completion time (FCT). We plot the CDF of FCT in Figure 4(c). HCC reduces the mean FCT by $62\%$, $54\%$, $42\%$ and the $99^{th}$ percentile FCT by $68\%$, $39\%$, and $49\%$ compared to NewReno, Cubic, and DCTCP, respectively. We note that the main goal of HCC is to run explicit max-min fairness in a distributed manner to achieve fast convergence and optimal fairness. This means that we do not have any special scheduling policy to prioritize short flows; still, we get relatively better results in terms of FCT since enforcing explicit max-min fairness keeps queue occupancy very low.

**Queue size.** Figure 4(d) illustrates the queue size of the bottleneck link sampled every 1 millisecond. HCC maintains a steady queue size very close to zero whereas NewReno and Cubic reach the full queue size of 1000 packets most of the time. DCTCP queue size stays around 65 packets which is the suggested value for the threshold of the queue policy [5]. This is a significant achievement for HCC to get almost $100\%$ utilization with no queuing using distributed max-min fairness with low communication overhead.

**Fairness.** An obvious result of using max-min fairness is major improvements in terms of fairness among flows even in short periods of time. Figure 4(e) shows the Jain fairness index calculated every 1 millisecond. Jain fairness index [34]




(a) (b) (c) (d)

Fig. 5. Comparison of HCC with TCP variants in terms of throughout, flow completion time, queue size, and packet latency for on-off gRPCs workload.

is always between 0 and 1 and values close to 1 indicate better fairness among a set of flows. For HCC, Jain fairness index is higher than 0.8 in 95% of the time and higher than 0.9 in 89% of the time. The index is significantly worse for others with the average below 0.5. In fact, other protocols can converge to high fairness indexes for longer flows in larger periods of time, but HCC can realize fairness even for short flows. Having said that, other metrics such as FCT and packet latency (presented below) are more important for short flows.

**Packet latency.** As a result of running distributed explicit max-min fairness, flows do not over-utilize the link bandwidths and do not create any queues, keeping the packet latency extremely small (close to the physical latency of the paths). Figure 4(f) shows the CDF of packet latency for all protocols and confirms extremely small packet latencies for HCC.

**Overhead.** HCC achieves high throughput, high fairness, small queue size, and small latency, using explicit congestion signals. Here, we compare the communication and computation overheads of HCC compared to a centralized entity that periodically collects flow demands, calculates max-min fair allocations, and distributes the allocations. We note that, besides the significant reduction in the overheads (as discussed below), HCC's local presence close to flow sources results in fast reaction at the edge compared to any centralized solution.

We compare the overhead of the centralized explicit congestion control with HCC in terms of total size of received and sent messages in Figures 4(g) and 4(h). The red bars show the overheads for the centralized solution. Besides, we have three variants of HCC with different Cycle values (values are relative to the link latencies of the associated level of agents). For each HCC variant, we show the overheads for the root rate allocation agent ($L3$, associated with depth 0 of the flow aggregation hierarchy) and the rate allocation agents associated with depth 1 of the flow aggregation hierarchy ($L2$). We do not present the agents with depth 2 as they are physically located at the end-hosts and can communicate with flows originated at the same machine without any overhead to the network.

HCC-2 is the variant used in all experiments reported up to this point. HCC-2 reduces the communication overhead to the core by 73% and the communication overhead from the core by 99.5% compared to the centralized solution. The quantized values of individual flow demands should be sent to the root agent to run the max-min fairness; however, on the reverse direction, it only needs to distribute a single value (Share *i.e.* total allocation of the child) to each of its children. This explains why HCC significantly reduces the sent overhead (Figure 4(h)). For the received overhead (Figure 4(g)), although the sum of the overhead of all agents is higher than the overhead of the centralized entity, the computation overhead is distributed among different physical agents and the imposed overhead on the core agent is significantly reduced. Also, the messages for HCC only traverse between their associated agents whereas for the centralized solution they should go all the way from a source of a flow to the centralized entity.

HCC-1, HCC-2, and HCC-3 in Figures 4(g) and 4(h), present effects of changing the Cycle values of rate allocation agents. As we decrease the Cycle of $L2$ and $L3$ rate allocation agents, moving from HCC-1 to HCC-2, and from HCC-2 to HCC-3, both the received and sent overheads are increased. On the other hand, the smaller the Cycle values, the more quickly HCC reacts to updates and the more accurate the distributed max-min fairness allocation will be. In our experiments, both HCC-1 and HCC-2 show near-optimal performance with extremely low overheads, but HCC-3 results in larger overheads. Furthermore, we study the effect of quantization by disabling quantization for HCC-2 which caused a 70% increase in the received overhead at the core and no significant changes in the sent overhead (yellow bars in Figures 4(g) and 4(h)).

### B. *On-off gRPCs workload*

We test HCC on a traffic pattern [12] that consists of a series of request and response gRPCs among clients and servers in a production network. The request payload is $196B$ and the response payload, which is the source of the congestion in the network, is $256KB$. Initially, we generate 500 flows. We assume each flow is started with a request gRPC from a random end-host on the right towards a random end-host on the left triggering a response gRPC in the reverse direction (the on period). When the client receives the entire response, it waits for $t_{pause}$ (the off period), which is a sample of a normal random variable with a mean of 2 milliseconds and a variance of 0.01 millisecond. Then, it sends a new request gRPC to the associated server and waits for its response. We continue this process for 0.5 second which is the total simulation time. The link latencies are similar to the previous set of experiments. The link capacities are set to $B_{core} = B_{agg} = B_{edge} = 10Gbps$ and we keep these capacities fixed throughout the experiment.

What differentiates this scenario from the previous one is the burstiness in the workload. A huge number of response gRPCs are triggered around the same time, immediately increasing the corresponding flow demand rates from zero to the line rate. As a result, despite the max-min fair rate allocation in HCC, we observe a burst of packets overwhelming the queue. To resolve the issue of bursty workloads in HCC, we send the

first packet of a flow after a rate update with a uniform delay between 0 and the inverse of the new allocated rate.

Figures 5(a) shows that the total throughput of flows can quickly converge to the maximum capacity and stays very stable as opposed to other protocols. Figures 5(b) presents the CDF of completion time of gRPC responses. HCC reduces the mean FCT by $47\%$, $46\%$, $9\%$ and the $99^{th}$ percentile FCT by $40\%$, $39\%$, and $6\%$ compared to NewReno, Cubic, and DCTCP, respectively. Furthermore, HCC maintains a near zero queue size in contrary to the other protocols as shown in Figure 5(c) and packet latency extremely close to physical path latency as shown in Figure 5(d).

## V. Conclusion and future work

In this paper, we present HCC, a practical solution for aggregating flows in a network. The objective is to control rate allocation in a collective and hierarchical manner, leveraging the network structure and flow correlations to reduce the control overhead. Our experiments show HCC can converge to the maximum utilization in very short time, and can result in an extremely high fairness index with a reasonable overhead. Interesting future directions are to use HCC for fairly distributing implicit congestion signals among groups of flows and to explore HCC for flow cooperation in networks that are not under a single authority.

## References


[1] V. Jacobson, "Congestion avoidance and control," in *ACM SIGCOMM computer communication review*, vol. 18, no. 4. ACM, 1988, pp. 314–329.

[2] M. Dong, Q. Li, D. Zarchy, P. B. Godfrey, and M. Schapira, "PCC: Re-architecting congestion control for consistent high performance." in *NSDI*, vol. 1, no. 2.3, 2015, p. 2.

[3] S. Abbasloo, C.-Y. Yen, and H. J. Chao, "Classic meets modern: A pragmatic learning-based congestion control for the internet," 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*, 2020, pp. 632–647.

[4] 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 Computer Communication Review*, vol. 45, no. 4. ACM, 2015, pp. 537–550.

[5] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan, "Data center TCP (DCTCP)," *ACM SIGCOMM computer communication review*, vol. 41, no. 4, pp. 63–74, 2011.

[6] N. Cardwell, Y. Cheng, C. S. Gunn, S. H. Yeganeh, and V. Jacobson, "BBR: Congestion-based congestion control: Measuring bottleneck bandwidth and round-trip propagation time," *Queue*, vol. 14, no. 5, pp. 20–53, 2016.

[7] 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*, 2020, pp. 514–528.

[8] J. Perry, A. Ousterhout, H. Balakrishnan, D. Shah, and H. Fugal, "Fastpass: A centralized zero-queue datacenter network," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 4, pp. 307–318, 2015.

[9] M. Ghobadi, S. H. Yeganeh, and Y. Ganjali, "Rethinking end-to-end congestion control in software-defined networks," in *Proceedings of the 11th ACM Workshop on Hot Topics in networks*. ACM, 2012, pp. 61–66.

[10] S. Floyd, T. Henderson, A. Gurtov *et al.*, "The NewReno modification to TCPs fast recovery algorithm," 1999.

[11] S. Ha, I. Rhee, and L. Xu, "CUBIC: a new TCP-friendly high-speed TCP variant," *ACM SIGOPS operating systems review*, vol. 42, no. 5, pp. 64–74, 2008.

[12] "Distbench test workloads," https://github.com/google/distbench/blob/main/workloads/README.md.

[13] J. Perry, H. Balakrishnan, and D. Shah, "Flowtune: Flowlet control for datacenter networks," in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, 2017, pp. 421–435.

[14] B. Vamanan, J. Hasan, and T. Vijaykumar, "Deadline-aware datacenter tcp (d2tcp)," *ACM SIGCOMM Computer Communication Review*, vol. 42, no. 4, pp. 115–126, 2012.

[15] C. Wilson, H. Ballani, T. Karagiannis, and A. Rowtron, "Better never than late: Meeting deadlines in datacenter networks," *ACM SIGCOMM Computer Communication Review*, vol. 41, no. 4, pp. 50–61, 2011.

[16] M. Alizadeh, S. Yang, M. Sharif, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker, "pfabric: Minimal near-optimal datacenter transport," *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4, pp. 435–446, 2013.

[17] W. Bai, L. Chen, K. Chen, D. Han, C. Tian, and H. Wang, "PIAS: Practical information-agnostic flow scheduling for commodity data centers," *IEEE/ACM Transactions on Networking*, vol. 25, no. 4, pp. 1954–1967, 2017.

[18] A. Munir, G. Baig, S. M. Irteza, I. A. Qazi, A. X. Liu, and F. R. Dogar, "Friends, not foes: synthesizing existing transport strategies for data center networks," in *Proceedings of the 2014 ACM conference on SIGCOMM*, 2014, pp. 491–502.

[19] D. Zhuo, Q. Zhang, V. Liu, A. Krishnamurthy, and T. Anderson, "Rack-level congestion control," in *Proceedings of the 15th ACM Workshop on Hot Topics in Networks*. ACM, 2016, pp. 148–154.

[20] D. Katabi, M. Handley, and C. Rohrs, "Congestion control for high bandwidth-delay product networks," in *Proceedings of the 2002 conference on Applications, technologies, architectures, and protocols for computer communications*, 2002, pp. 89–102.

[21] N. Dukkipati, *Rate Control Protocol (RCP): Congestion control to make flows complete quickly*. Citeseer, 2008.

[22] L. Jose, S. Ibanez, M. Alizadeh, and N. McKeown, "A distributed algorithm to calculate max-min fair rates without per-flow state," *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 3, no. 2, pp. 1–42, 2019.

[23] I. Cho, K. Jang, and D. Han, "Credit-scheduled delay-bounded congestion control for datacenters," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, 2017, pp. 239–252.

[24] A. Demers, S. Keshav, and S. Shenker, "Analysis and simulation of a fair queueing algorithm," *ACM SIGCOMM Computer Communication Review*, vol. 19, no. 4, pp. 1–12, 1989.

[25] A. K. Parekh and R. G. Gallager, "A generalized processor sharing approach to flow control in integrated services networks: the single-node case," *IEEE/ACM transactions on networking*, vol. 1, no. 3, pp. 344–357, 1993.

[26] Z. Yu, J. Wu, V. Braverman, I. Stoica, and X. Jin, "Twenty years after: Hierarchical core-stateless fair queueing," in *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, 2021, pp. 29–45.

[27] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," *ACM SIGCOMM computer communication review*, vol. 38, no. 4, pp. 63–74, 2008.

[28] S. Ketabi and Y. Ganjali, "Hierarchical congestion control (HCC): Cooperation of uncorrelated flows for better fairness and throughput," in *NOMS 2020-2020 IEEE/IFIP Network Operations and Management Symposium*. IEEE, 2020, pp. 1–5.

[29] S. Ketabi, M. Buckley, P. Pazhooheshy, F. Farahvash, and Y. Ganjali, "Correlation-aware flow consolidation for load balancing and beyond."

[30] D. C. Montgomery and G. C. Runger, *Applied statistics and probability for engineers*. John Wiley & Sons, 2010.

[31] S. Ketabi and Y. Ganjali, "Perfect is the enemy of good: Lloyd-max quantization for rate allocation in congestion control plane," in *NOMS 2020-2020 IEEE/IFIP Network Operations and Management Symposium*. IEEE, 2020, pp. 1–9.

[32] B. Awerbuch and Y. Shavitt, "Converging to approximated max-min flow fairness in logarithmic time," in *INFOCOM'98. Seventeenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, vol. 3. IEEE, 1998, pp. 1350–1357.

[33] G. F. Riley and T. R. Henderson, "The ns-3 network simulator," in *Modeling and tools for network simulation*. Springer, 2010, pp. 15–34.

[34] R. Jain, A. Durresi, and G. Babic, "Throughput fairness index: An explanation," in *ATM Forum contribution*, vol. 99, no. 45, 1999.