DESIGNING DATACENTER TRANSPORTS FOR LOW LATENCY AND HIGH THROUGHPUT

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF ELECTRICAL ENGINEERING
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

BEHNAM MONTAZERI NAJAFABADI
JUNE 2019

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

John Ousterhout, Primary Adviser

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Christos Kozyrakis

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Mendel Rosenblum

Approved for the Stanford University Committee on Graduate Studies.

Patricia J. Gumport, Vice Provost for Graduate Education

Abstract

Recent trends in datacenter computing have created new operating conditions for network transport protocols. One of these trends is applications that operate at extreme low latency. Modern datacenter networking hardware offers the potential for very low latency communication; round-trip times of 5µs or less are now possible for short messages. A variety of applications have arisen that can utilize this low latency because their workloads are dominated by very short messages (a few hundred bytes or less); Facebook's Memcached [21] and RAMCloud [27] storage system are two instances of such low latency applications.

Transport protocols, however, are traditionally designed to achieve high throughput rather than low latency. They sometimes even sacrifice low latency for the sake of achieving high throughput; TCP and TCP-like transports, which are the de facto transports used in most applications today are examples of high throughput transports. These transports are known to impose high latency for short messages when the network load is high. That's because their sender driven congestion control mechanism creates large queues in the network and short messages can experience long head-of-line blocking delays in these queues. So even though these large queues can bring high throughput and bandwidth utilization, they can extremely hurt short message latencies.

In this thesis we postulate that low latency and high throughput are not mutually exclusive goals for transport protocols. In fact we design a new transport protocol named Homa that can achieve both of these goals in datacenter networks. Homa [23] is a transport protocol for datacenter networks that provides exceptionally low latency, especially for workloads with a high volume of very short messages, and it also supports large messages and high network bandwidth utilization.

Homa uses in-network priority queues to ensure low latency for short messages; priority allocation is managed dynamically by each receiver and integrated with a receiver-driven flow control mechanism. Homa also uses controlled overcommitment of receiver downlinks to ensure efficient bandwidth utilization at high load. We evaluate Homa in both simulations and a real system implementation. Our implementation of Homa delivers 99th percentile round-trip latencies less than

15 μs for short messages on a 10 Gbps network running at 80% network load. These latencies are almost 100x lower than the best published measurements of an implementation in prior works. In simulations, Homa's latency is roughly equal to pFabric [5] and significantly better than pHost [14], PIAS [7], and NDP [16] for almost all message sizes and workloads. Homa can also sustain higher network loads than pFabric, pHost, or PIAS.

Preface

The simulator used to evaluate Homa for this dissertation is available at `https://github.com/PlatformLab/HomaSimulation`. Homa's implementation in RAMCloud is available at `https://github.com/PlatformLab/RAMCloud`.

Acknowledgments

Getting through the PhD program would have been a lot more difficult for me if there wasn't the support and friendship I received from many people. The pages of my appreciation and gratefulness for these individuals can itself turn into a large book. But here I have no choice but to try and condensate my appreciation in the next few paragraphs.

I've been truly privileged to have the opportunity to do my PhD at Stanford University and spend time with some of the most talented and hardworking individuals in the world. I'd like to acknowledge some of these individuals here for the help and support they provided to me.

First I like to express my gratitude to my PhD adviser, John Ousterhout. John is probably the most on time, well organized, and curious person I know. He is an outstanding character with immense enthusiasm about high quality research. His dissatisfaction with superficial research made me a better researcher. So many times when I presented the results of my work to him, he asked me to dig deeper and measure many layers below the surface to truly understand why and how the system behaves the way it does. I learned from John to look for the ways that I can redesign and improve a system to the extent that it can't be improved any further with today's technology; he taught me to not just shoot for the better, but for the best in my research. John's method of leadership in the RAMCloud group and his hands-on style of advising made my PhD experience a pleasant one. I'd like to thank John for believing in me and supporting me to grow during my PhD and become the researcher and the person I am today. I hope I can continue to use his guidance and advice to even further grow in the future.

I like to thank the past and present members of the RAMCloud group. The previous generation of the group: Ankita Kejriwal, Diego Ongaro, Ryan Stutsman, and Steve Rumble. They built many critical parts of RAMCloud version 1.0 and paved the way for me and the rest of the second generation students in the group to find interesting research problems through experimentation with RAMCloud. The second generation of the RAMCloud group, they are my partners in crime throughout my PhD and I like to formally thank them: Jonathan Ellithorpe, Collin Lee, Yilong Li, Seo Jin Park, Stephen Yang, and Henry Qin. Thanks for endless number of hours that they spent thinking about my problems, helping me to debug my programs when I was stuck, and listening to me presenting my research. Among all the RAMCloud group's students, I like to specially thank Yilong Li. While I was busy designing and implementing my ideas in the simulator, he helped me by spearheading the implementation of my design in the RAMCloud's code base and evaluating it. Without his help, the quality of my research would have significantly reduced.

Thanks to Mohammad Alizadeh for the time he spent with me both as a mentor and more importantly as a close friend. Mohammad was like a co-adviser to me during my PhD work. His deep understanding of datacenter networks was a great source of learning for me throughout my PhD. I thank him for all the support, friendship and help that he provided me throughout my PhD at Stanford. It's been a great pleasure for me to know him and become a friend with him.

I also like to thank my PhD examination committee: Mendel Rosenblum, Nick McKeown, Christos Kozyrakis and my committee chair Amin Savafi-Naeini. Their sound advice and feedback on my dissertation work made it a better work. Mendel with all of his spontaneous and incredibly thought provoking questions he asks, Christos with his extensive views on the top to bottom of the software stack, and Nick with his ability to find different ways that a system can fail and not perform well, they all made me think deeper about my work and shape it to more robust work. I should also thank Amir for making time for me to be the chair of my PhD oral examination committee.

I want to thank various organizations, agencies and companies that directly or indirectly funded my research. This dissertation was made possible by supports from C-FAR (one of six centers of STARnet, a Semiconductor Research Corporation program, sponsored by MARCO and DARPA) and by the industrial affiliates of the Stanford Platform Laboratory, including VMWare, Google, Huawei, NEC, Cisco, etc.

Finally, I'd like to thank my family for their unconditional love and support that they provided me. My family has always been the most important part my life and I'm ethereally grateful to them and appreciate them beyond any measures. First my mother, my father, my sister Behnaz, and my brothers Behshad and Behrad. They always believed in me no matter what, they put the courage in me to pursue my dreams, they stood behind me whenever I felt the pressure is surmounting me. And last but not least, my beloved wife and the love of my life Azar, and my cute little daughter Kimiya. Thank you Azar for being with me during all my PhD years. You have fueled my courage and my ambition and I cannot imagine a better partner to navigate the life with. Kimiya, you are the joy of life and I hope I can be a great dad to you. To my family, I love you all with my heart.

To my wife Azar and my daughter Kimiya.
To my parents Bahram and Mahin, my sister Behnaz, and
my brothers Behshad and Behrad.

Contents

List of figures

Chapter 1

Introduction

The rise of datacenter computing over the last decade has created new opportunities and challenges for network protocols. On one hand, modern datacenter networking hardware offers the potential for very low latency communication. Round-trip times of 5 µs or less are now possible for short messages, and a variety of applications have arisen that can take advantage of this latency [21, 30, 27]. In addition, many datacenter applications use request-response protocols that are dominated by very short messages (a few hundred bytes or less).

On the other hand, existing transport protocols are ill-suited to these conditions, so the latency they provide for short messages is far higher than the hardware potential, particularly under high network loads. Almost all existing applications rely on the TCP protocol for network transport, and TCP was not designed for this environment. For example, it cannot identify messages within a flow, so it cannot give priority to smaller messages, and it depends on buffer occupancy for congestion control, which increases latency under load. As a result, most applications today cannot reap the full benefits of low latency datacenter networks.

Recent years have seen numerous proposals for better transport protocols, including improvements to TCP [3, 4, 35] and a variety of new protocols [36, 18, 5, 28, 15, 7, 16]. However, each of these proposals suffers from limitations. For example, pFabric [5] offers near-optimal latency by prioritizing small messages, but it requires specialized networking hardware, which limits its adoption. Fastpass [15] eliminates queuing by scheduling messages centrally, but it adds an extra round-trip delay, which is a severe penalty for short messages.

Moreover, none of these designs considers today's small message sizes; they are based on heavy-tailed workloads where 100 Kbyte messages are considered "small," and latencies are often measured in milliseconds, not microseconds. As a result, there is still no practical solution that provides near-hardware latencies for short messages under high network loads. For example, we know of no existing implementation with tail latencies of 100 µs or less at high network load (within 20x of the hardware potential).

In this dissertation we revisit the challenge of creating a low latency transport for datacenters. Our assumptions differ from previous work in two key respects: first, we assume that workloads are dominated by very short messages (a few hundred bytes or less); and second, we assume the availability of ultra-low-latency networks (5 µs round trip times). Each of these assumptions creates new challenges. For example, we will show that a high volume of short messages precludes centralized solutions and requires the use of network priorities for preemption. The 5 µs end-to-end latency means that seemingly small inefficiencies such as the non-preemptibility of partially transmitted packets can have a significant impact on performance.

Homa is the name of a new transport protocol we designed to achieve low latency for small messages in datacenter environments, even in the presence of high network loads and competing large messages. Our implementation of Homa achieves 99th percentile round trip latencies less than 15 µs for small messages at 80% network load with 10 Gbps link speeds, and it does this even in the presence of competing large messages. Across a wide range of message sizes and workloads, Homa achieves 99th percentile latencies at 80% network load that are within a factor of 2–3.5x of the minimum possible latency on an unloaded network. Although Homa favors small messages, it also improves the performance of large messages in comparison to TCP-like approaches based on fair sharing.

In our simulations, across various workloads and even at 80% network load, Homa provides 99th-percentile tail latencies for short messages within a factor of 2.2x of an unloaded network. pFabric's performance is considered near-optimal, and at high loads Homa's short message latencies are equal or sometimes up to 30% lower than pFabric on heavy-tailed workloads. Homa can also sustain 2–17% higher network loads than pFabric, and it can be implemented without any modifications to networking hardware.

Homa uses three key innovations to achieve its performance:

1. Priorities. Homa makes aggressive use of the priority queues that are provided by modern network switches. The number of these priority queues is typically limited in commodity network fabrics. So in order to make the most of this limited number, Homa assigns packet priorities dynamically at receivers. Homa takes advantage of these priorities to implement SRPT (shortest remaining processing time first). Priorities allows shorter messages to preempt longer ones quickly, thereby improving latency. In Homa, each receiver independently

determines the priorities for its incoming packets.

2. Receiver-based scheduling. Unlike TCP which uses a sender-driven congestion control mechanism, Homa implements a receiver-driven flow control mechanism like pHost [14] and NDP [16]. In datacenter networks, we expect congestion to occur primarily at the downlinks from the top-of-rack switches to the receivers. Therefore, Homa manages congestion from the receiver since the receiver is the closest end-point to the congestion; receivers schedule incoming traffic on a packet-by-packet basis. The priority assignment mechanism is integrated with the congestion control mechanism to achieve optimal low latency for short messages. Homa improves tail latency by 2–16x compared to previous receiver-driven approaches. In comparison to sender-driven priority mechanisms such as PIAS [7], Homa provides a better approximation to SRPT; this reduces tail latency by 0–3x over PIAS.

3. Controlled overcommitment. Homa's third contribution is its use of Controlled Overcommitment, where a receiver allows a few senders to transmit simultaneously. Slightly overcommitting receiver downlinks in this way allows Homa to use network bandwidth efficiently: Homa can sustain network loads 2–33% higher than pFabric [5], PIAS, pHost, and NDP. Homa limits the overcommitment and integrates it with the priority mechanism. The priority mechanism is key to Homa's ability to operate at high network loads; it allows receivers to overcommit their downlinks, which is necessary to achieve high bandwidth utilization, while still ensuring low latency for short messages; the priority mechanism prevents queuing of short messages.

Homa has several other unusual features that contribute to its high performance. It uses a message-based architecture rather than a streaming approach; this eliminates head-of-line blocking at senders and reduces tail latency by 100x over streaming transports such as TCP. Homa is connectionless, which reduces connection state in large-scale applications. It has no explicit acknowledgments, which reduces overheads for small messages, and it implements at-least-once semantics rather than at-most-once.

The rest of this dissertation is organized as follows: In Chapter 2, we present the motivation and rationale for the design of Homa and describe the key ideas behind our design. In Chapter 3, we focus on a simplified many-to-one (i.e. many senders, single receiver) traffic pattern and discuss the design of Homa's receiver-driven congestion control and priority assigns scheme. In Chapter 4, we evaluate the design of Chapter 3, using a packet simulator; this chapter is heavily focused on the evaluation of the priority assignment mechanism. In Chapter 5, we arrive at the complete design of

Homa's transport scheme under many-to-many traffic patterns; we discuss the shortcomings of the transport we designed and evaluated in the previous two chapters, then we extend that design with the overcommitment mechanism to address those shortcomings. In Chapter 6, we evaluate the complete design of Homa; we compare Homa against state of the art transports using packet-level simulators and demonstrate how Homa outperforms other practical low latency transports. We present a system implementation of Homa in Chapter 7 and evaluate it under realistic workloads in datacenter environments. We discuss related works in Chapter 8 and Homa's limitations in Chapter 9. Finally, we close the dissertation with a few concluding remarks in Chapter 10.

Chapter 2

Motivation and Key Ideas

The primary goal of Homa is to provide the lowest possible latency for short messages at high network load using commodity networking hardware. In the course of this dissertaion, we focus on tail message latency (99th percentile), as it is the most important metric for datacenter applications [3, 37]. A large body of work has focused on low latency datacenter transport in recent years. However, as our results show, existing designs are sub-optimal for tail latency at high network load, particularly in networks with raw hardware latency in the single-digit microseconds [32, 10, 38, 22]. In this chapter, we discuss the challenges that arise in such networks and we derive Homa's key design features. These features include dynamic allocation of in-network priorities, receiver-driven congestion control, and limited use of blind transmissions.

2.1 Motivation: Tiny Latency for Tiny Messages

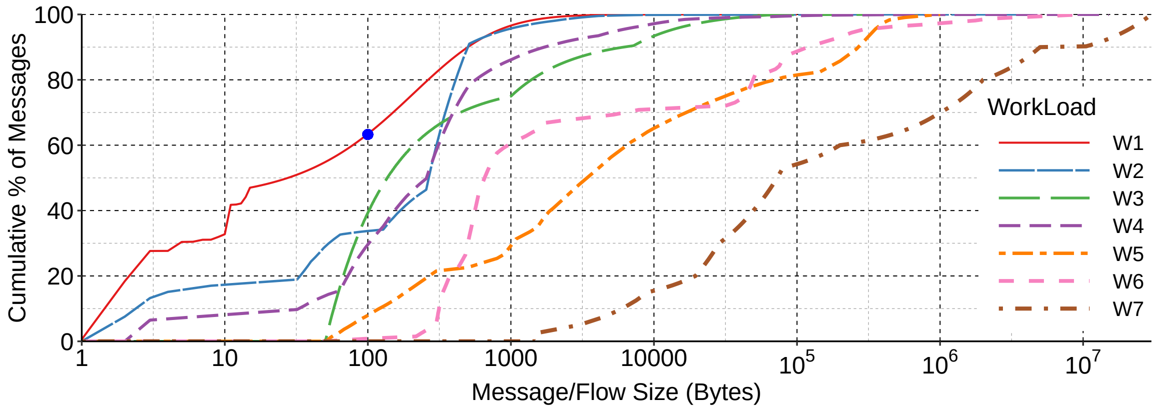
State-of-the-art cut-through switches have latencies of at most a few hundred nanoseconds [34]. Low latency network interface cards and software stacks (e.g., DPDK [10]) have also become common in the last few years. These advances have made it possible to achieve one-way latencies of a few microseconds in the absence of queuing, even across a large network with thousands of servers (e.g., a 3-level fat-tree network).

We focus on message latency (not packet latency) since it reflects application performance. A message is a block of bytes of any length transmitted from a single sender to a single receiver. The sender must specify the size of a message when presenting its first byte to the transport, and the receiver cannot act on a message until it has been received in its entirety. Knowledge of message sizes is particularly valuable because it allows transports to prioritize shorter messages. We assert that a message-based transport can be used to implement streams like those provided by traditional sockets, and it can also be implemented underneath a socket interface (e.g. by guessing message boundaries), but we leave the details to future work.

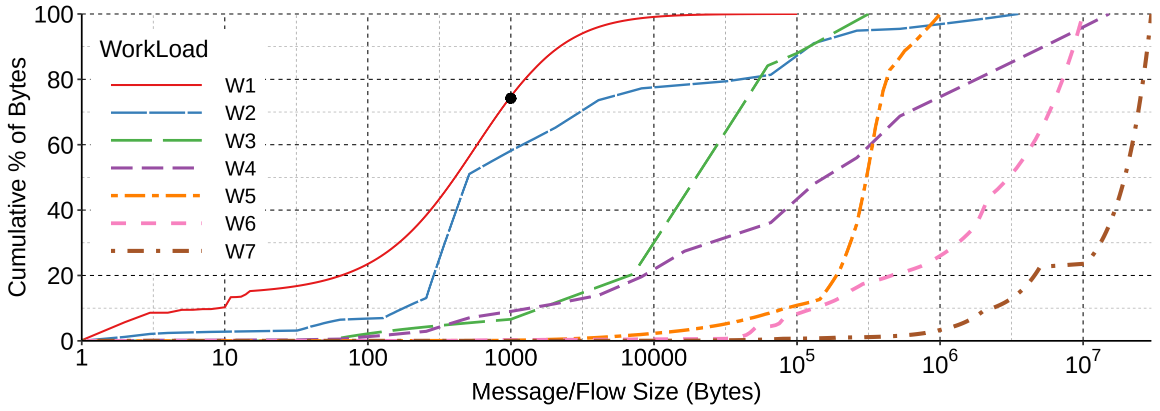
Many datacenter applications rely on request-response protocols with tiny messages of a few hundred bytes or less. In typical remote procedure call (RPC) use cases, it is almost always the case that either the request or the response is tiny, since data usually flows in only one direction. The data itself is often very short as well. Figure 2.1 shows a collection of workloads that we used to design and evaluate Homa, most of which were measured from datacenter applications at Google, Facebook, and Microsoft. In three of these workloads, more than 85% of messages were less than 1000 bytes. In the most extreme case (W1), more than 70% of all network traffic, measured in bytes, was in messages less than 1000 bytes.

To our knowledge, almost all prior work has focused on workloads with very large messages. For example, in the Web Search workload used to evaluate DCTCP [3] and pFabric [5] (W7 in Figure 2.1), messages longer than 1 Mbyte account for 95% of transmitted bytes, and any message shorter than 100 Kbytes was considered “short”. Most subsequent work has used the same workloads. Unfortunately, the method used to measure these workloads, significantly overestimates message sizes: to obtain these workloads, message sizes were estimated from packet captures based on inactivity of TCP connections; connections are distinguished by 5-tuple descriptors in the packet headers and a message (i.e. flow in previous works) and its size is identified if a connection remains silent beyond a threshold (e.g. 50 ms). Since application-level message boundaries aren’t defined in a TCP connection, a TCP flow can contain many (sometimes up to tens of) closely-spaced messages. Our definition of a message (given previously in this section) is very different from a traditional 5-tuple flow, in the sense that a message is measured from application-level traces and is a much finer grain boundary between blocks of transmitted data. In Figure 2.1, workloads W1–W4 were measured explicitly in terms of application-level messages, and they show much smaller sizes than workloads W5–W7, which were extracted from packet captures.

Almost all existing and proposed transport protocols follow TCP’s example and implement connection-oriented byte streams. Unfortunately this results in head-of-line-blocking, where a short message for a given destination is queued behind a long message for the same destination. Chapter 7 shows that this increases tail latency by 100x for short messages. Most recent proposals, such as DCTCP, pFabric, and PIAS, assume dozens of connections between each source-target pair, so that each message has a dedicated connection. However, this approach results in an explosion of



(a)



(b)

W1	Accesses to a collection of Memcached servers at Facebook, as approximated by the statistical model of the ETC workload in Section 5 and Table 5 of [6].
W2	Measured from the search application workload at Google [33].
W3	A synthesized workload.
W4	Aggregated workload from all applications running in a Google datacenter [33].
W5	Measured from intra cluster traffic of Web Server applications at Facebook [31].
W6	Aggregated workload from all Hadoop traffic at Facebook [31].
W7	Web search workload used for DCTCP, pFabric, pHost, NDP, PIAS, etc. [3], [5], [16], [14], [7].

(c)

Figure 2.1: The workloads used to design and evaluate Homa. The upper graph shows the cumulative distribution of message sizes weighted by number of messages. The bottom graph is a cumulative distribution of messages weighted by bytes. The workload names W1 to W7 are lexicographically ordered by average message size: W1 has the smallest average message size, and W7 has the largest average message size (i.e. the most heavy-tailed workload)

connection state. Even a single connection for each application-server pair is problematic for large-scale applications ([24] §3.1, [12] §3.1), so it is not realistic to use multiple connections. Homa solves these problems with a lightweight message-oriented mechanism.

Unfortunately, existing datacenter transport designs cannot achieve the lowest possible latency for tiny messages at high network load. We explore the design space in the next section, but consider, for example, designs that do not take advantage of in-network priorities (e.g., HULL [4], PDQ [18], NDP [16]). These designs attempt to limit queue buildup, but none of them can eliminate queuing altogether. The state-of-the-art approach, NDP [16], strictly limits queues to 8 packets, equivalent to roughly 10 μs of latency at 10 Gbps. While this queuing latency has negligible impact in a network with moderate latency (e.g., RTTs greater than 50 µs) or for moderately-sized messages (e.g., 100 KBytes), it increases by 5x the completion time of a 200-byte message in a network with 5 μs RTT.

2.2 The Design Space

We now present a walk through the design space of low latency datacenter transport protocols. We derive Homa's four key design principles: (i) transmitting short messages blindly, (ii) using in-network priorities, (iii) allocating priorities dynamically at receivers in conjunction with receiver-driven rate control, and (iv) controlled overcommitment of receiver downlinks. While some past designs use the first two of these techniques, in this dissertation, we show that combining all four techniques is crucial to deliver the lowest levels of latency at high network load.

The key challenge in delivering short messages with low latency is to eliminate queuing delays. Similar to prior work, we assume that bandwidth in the network core is sufficient to accommodate the offered load, and that the network supports efficient load-balancing [11, 17, 2], so that packets are distributed evenly across the available paths (we assume simple randomized per-packet spraying in our design). As a result, queueing occurs primarily in the downlinks from top-of-rack switches (TORs) to machines. This happens when multiple senders transmit simultaneously to the same receiver. The worst-case scenario is incast, where an application initiates RPCs to many servers concurrently and the responses all arrive at the same time.

Before we dive into the design principles that we derived Homa from, let's take a step back and describe the transport problem, At a high level, a datacenter can be modeled as a bunch of servers that are connected to a big switch. Each server can act as a sender of some messages and/or a receiver for a some other messages. In general, multiple senders have messages to transmit to multiple receivers. The message is presented to the sending transport in its entirety (i.e., its size is known), and the receiver cannot act on the message until it has been received in its entirety. We assume that messages are transmitted as part of an ongoing connection that can maintain state; but we do not include the cost of connection setup in the message latency.

There is no time to schedule every packet. An ideal scheme might attempt to schedule every packet at a central arbiter, as in Fastpass [28]. Such an arbiter could take into account all the messages and make a global scheduling decision about which packet to transmit from each sender and when to transmit it. The arbiter could in theory avoid queues in the network altogether. However, this approach triples the latency for short messages: a tiny, single-packet message takes at least 1.5 RTTs if it needs to wait for a scheduling decision, whereas it could finish within 0.5 RTT if transmitted immediately. Receiver-based scheduling mechanisms such as ExpressPass [9] suffer the same penalty.

In order to achieve the lowest possible latency, short messages must be transmitted blindly, without considering potential congestion. In general, a sender must transmit enough bytes blindly to cover the round-trip time to the receiver (including software overheads on both ends); during this time the receiver can return explicit scheduling information to control future transmissions, without introducing additional delays. We refer to this amount of data as RTTbytes; it is about 10 KB in our implementation of Homa for 10 Gbps networks.

Buffering is a necessary evil. Blind transmissions mean that buffering can occur when multiple senders transmit to the same receiver. No protocol can achieve minimum latency without incurring some buffering. But, ironically, when buffering occurs, it increases latency. Many previous designs have attempted to reduce buffering, e.g., with carefully-engineered rate control schemes [3, 38, 22], reserving bandwidth headroom [4], or even strictly limiting the buffer size to a small value [16]. However, none of these approaches can completely eliminate the latency penalty of buffering.

NDP [16] reduces queue lengths significantly, to a maximum of eight jumbo packets per queue, but this still leaves queueing delays of almost 60 μs on a 10 Gbps network, a greater than 10x penalty in a network with a 5 μs round-trip time.

In-network priorities are a must. Given the inevitability of buffering, the only way to achieve the lowest possible latency is to use in-network priorities. Each output port in a modern switch supports a small number of priority levels (typically 8), with one queue for each priority. Each incoming packet indicates which queue to use for that packet, and output ports service higher priority queues before lower priority ones. The key to low latency is assigning packet priorities so that short messages bypass queued packets for longer messages.

This observation is not new; starting with pFabric [5], several schemes have shown that switch-based priorities can be used to improve message latency [15, 14, 7, 8]. These schemes use priorities to implement various message-size-based scheduling policies. The most common of these policies is SRPT (shortest remaining processing time first), which prioritizes packets from messages with the fewest bytes remaining to transmit. SRPT provides near-optimal average message latency, and as shown in prior work [18, 5], it also provides very good tail latency for short messages. Homa implements an approximation of SRPT (though the design can support other policies as well): if there are multiple packets ready to transmit on a link, Homa favors the packet whose message has the fewest bytes remaining to transmit. Our evaluation shows that this produces very good results for short messages, and that it outperforms other approaches that do not use SRPT, such as PIAS and NDP.

Unfortunately, in practice, no existing scheme can deliver the near-optimal latency of SRPT at high network load. pFabric approximates SRPT accurately, but it requires too many priority levels to implement with today's switches. PIAS [7] works with a limited number of priorities, but it assigns priorities on senders, which limits its ability to approximate SRPT. In addition, it works without message sizes, so it uses a "multi-level queue" scheduling policy. As a result, PIAS has high tail latency both for short messages and long ones. QJUMP [15] requires priorities to be allocated manually on a per-application basis, which is too inflexible to produce optimal latencies.

Making best use of limited priorities requires receiver control. The next question is how to allocate the limited priorities to packets of different messages. For heavy-tailed workloads, this is a relatively easy task, because one or two priorities are usually sufficient to isolate short messages from the very large messages that contribute the majority of the load. For example, in the Web Search workload (W5 in Figure 2.1), messages smaller than 100 Kbytes represent less than 1% of all transmitted bytes; we can schedule all these messages in a single high priority queue, dramatically reducing their latency. Hence performance is relatively insensitive to the priority scheme for such workloads, and prior work has demonstrated good results with few priorities and simple priority schemes [5, 15, 7, 8].

However, for workloads with very short message sizes, it becomes crucial to make the best use of available priorities. In particular, these workloads do not include huge, easily-identifiable messages which we can easily isolate from the short messages; instead, we must deal with a continuum of message sizes. Small changes to the priority assignments can have a large impact on latency for short messages. Implementing an accurate approximataion of SRPT can be a hard challenge when we have a limitted number of priorities.

To produce the best approximation of SRPT with only a small number of priority levels, the priorities should be determined by the receiver. Except for blind transmissions, the receiver knows the exact set of messages vying for bandwidth on its downlink from the TOR switch. As a result, the receiver can best decide which priority to use for each incoming packet. In addition, the receiver can amplify the effectiveness of the priorities by integrating them with a packet scheduling mechanism.

pHost [14], the closest prior scheme to Homa, is an example of using a receiver-driven approach to approximate SRPT. Its primary mechanism is packet scheduling: senders transmit the first RTTbytes of each message blindly, but packets after that are transmitted only in response to explicit grants from the receiver. Receivers schedule the grants to implement SRPT while controlling the influx of packets to match the downlink speed.

However, pHost makes only limited use of priorities: it statically assigns one high priority for all blind transmissions and one lower priority for all scheduled packets. This impacts its ability to approximate SRPT in two ways. First, it bundles all blind transmissions into a single priority. While this is reasonable for workloads where most bytes are from large messages (W5-W7 in Figure 2.1), it is problematic for workloads where a large fraction of bytes are transmitted blindly (W1-W4). Second, for messages longer than RTTbytes, pHost cannot preempt a larger message immediately for a shorter one. Once again, the root of the problem is that pHost bundles all such messages into a single priority, which results in queueing delays. We show in Chapter 3 that this creates preemption lag, which hurts latency, particularly for medium-sized messages that last a few RTTs.

Receivers must allocate priorities dynamically. Homa addresses pHost's limitations by dynamically allocating multiple priorities at the receivers. Homa uses a novel priority allocation scheme where each receiver dynamically controls the priorities of its incoming packets based on its observed workload and the exact set of messages destined to that receiver. Each receiver allocates priorities for its own downlink using two mechanisms. For messages larger than RTTbytes, the receiver communicates a priority for each scheduled packet to its sender dynamically based on the exact set of inbound messages. This eliminates almost all preemption lag. For short messages sent blindly, the sender cannot know about other messages inbound for the receiver. Even so, the receiver can provide guidance in advance to senders based on its recent workload. Our experiments show that dynamic priority management reduces tail latency considerably in comparison to static priority allocation schemes such as those in pHost or PIAS.

Receivers must overcommit their downlink in a controlled manner. Scheduling packet transmissions with grants from receivers reduces buffer occupancy, but it introduces a new challenge: a receiver may send grants to a sender that does not transmit to it in a timely manner. This problem occurs, for instance, when a sender has messages for multiple receivers; if more than one receiver decides to send it grants, the sender cannot transmit packets to all such receivers at full speed. This wastes bandwidth at the receiver downlinks and can significantly hurt performance at high network load. For example, we find that the maximum network load that pHost can support ranges between 58% and 73% depending on the workload, despite using a timeout mechanism to mitigate the impact of unresponsive senders. In contrast, Homa can support much larger maximum network loads between 89% and 92% depending on the workload (details in Chapter 6). NDP [16] also schedules incoming packets to avoid buffer buildup, and it suffers from a similar wasted bandwidth problem as in pHost.

To address this challenge, Homa's receivers intentionally overcommit their downlinks by granting simultaneously to a small number of senders; this results in controlled packet queuing at the receiver's TOR but is crucial to achieve high network utilization and the best message latency at high load (Chapter 5).

Senders need SRPT also. Queues can build up at senders as well as receivers, and this can result in long delays for short messages. For example, FIFO packet transmit queues in the NIC can result in head-of-line blocking and high tail latency for short messages. In Chapter 6 we demonstrate that this increases tail latency by 100x for short messages. In addition, most existing protocols implement byte streams, and an application typically uses a single stream for each destination. However, this can result in head-of-line-blocking, where a short message for a given destination is queued in the byte stream behind a long message for the same destination. In Chapter 7 we that this can also increases tail latency by 100x for short messages. For low tail latency, senders must ensure that short outgoing messages are not delayed by long ones, neither in FIFO transmit queues in the NIC nor within the streams of packets to the same destination.

Run-to-completion is essential for low latency. Most existing protocols, such as TCP and NDP [16], implement some form of fair sharing: if there are multiple messages inbound to a receiver, the receiver shares its downlink bandwidth among them so they all progress at the same rate. Unfortunately, this causes all of the messages to finish slowly. To minimize latency, the receiver should pick one message at a time and allocate the entire downlink bandwidth to that message until it completes. SRPT is an example of a run-to-completion protocol; as bytes of a message are received, its priority increases, so the message runs to completion unless a new shorter message appears.

SRPT scheduling is good for large messages as well as small ones. A potential concern with SRPT is that it might penalize large messages. However, we show that in comparison to other approaches, SRPT actually improves performance for large messages due to the "run-to-completion" behavior; once a message becomes highest priority, it tends to stay that way and grabs all the bandwidth until it completes. In contrast, approaches that produce fair sharing behavior, such as TCP, complete all messages slowly.

Thus, SRPT benefits all messages except a very small number of the largest messages under high network load; these messages can suffer long delays during bursts of activity. We hypothesize that there are simple techniques for mitigating these delays. One possibility is to dedicate a small fraction of the link bandwidth to the oldest message; eventually the remaining bytes of that message is going to drop to the point where SRPT gives it the full link bandwidth. We leave a full exploration of these techniques to future work.

2.3 Putting It All Together: Homa Overview

Figure 2.2 shows an overview of the Homa protocol. Homa delegates congestion control and priority allocation decisions to the receiver; since congestion occurs mostly on the receiver's downlink, the receiver is in the best position to make decisions (e.g., packet scheduling and priority allocations) for its own traffic based on the observed workload.

When a message arrives at a sender's transport module, Homa divides messages into two parts: an initial unscheduled portion followed by a scheduled portion. The sender transmits the unscheduled packets (RTTbytes of data) immediately, but it does not transmit any scheduled packets until instructed by the receiver. The arrival of an unscheduled packet makes the receiver aware of the message; the receiver then requests the transmission of scheduled packets by sending one grant packet for each scheduled packet. When the sender receives a grant, it transmits the packet indicated in the grant.

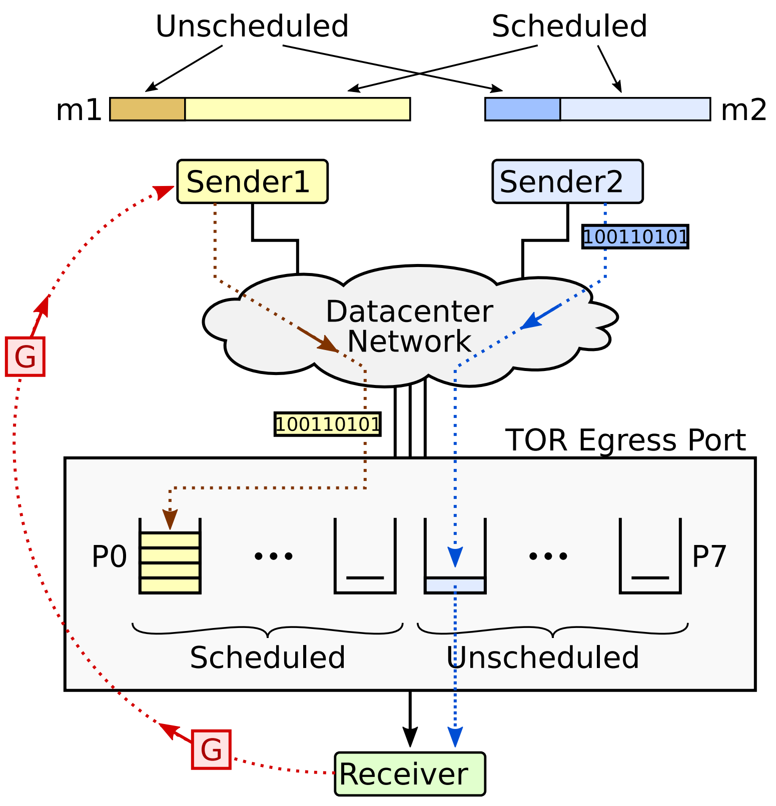


Figure 2.2: Overview of the Homa protocol. Sender1 is transmitting scheduled packets of message m1, while Sender2 is transmitting unscheduled packets of m2 Unscheduled packets have higher priority, so packets of m1 get queued in the TOR egress port.

Homa's receivers dynamically set priorities for scheduled packets and periodically notify senders of a set of thresholds for setting priorities for unscheduled packets. Each receiver decides independently how to use the available priorities in the TOR for unscheduled and scheduled packets; higher priorities are used for unscheduled packets and lower priorities for scheduled packets. The receiver computes a set of threshold values based on the workload and sends them to the sender to use to set priorities for unscheduled packets. For scheduled packets, the receiver dynamically decides a priority based on the real-time set of active messages for that receiver; it informs the sender about these priority decisions in grants for scheduled packets.

The priority in a packet is used not just for the final downlink, but throughout the lifetime of the packet as it traverses the network core. Since different receivers may assign priorities differently, it is possible that their priorities conflict in the core switches. The best way to handle this problem is to disable priorities for all ports except leaf downlinks. However, even if this isn't possible, queueing is rare in the core, so we don't expect priority conflicts to affect overall performance. We leave a detailed analysis of this issue to future work.

Finally, a receiver sometimes intentionally overcommits its downlink by granting simultaneously to multiple senders; this results in controlled packet queuing at the receiver's TOR but is crucial to achieve high network utilization (Chapter 5). The receivers implement controlled overcommitment to sustain high utilization in the presence of unresponsive senders.

Overall, the receiver uses a combination of grants and priority assignments (for unscheduled and scheduled packets). The net effect is a close approximation of the SRPT scheduling policy using a small number of priority queues. We demonstrate that SRPT provides excellent message latency and yields this excellent performance across a broad range of workloads and traffic conditions. Although we focus on SRPT for this thesis, Homa's receiver-driven approach is flexible enough to support a variety of other policies, including FIFO, round-robin, or fixed priorities for particular applications. We leave an exploration of alternative policies to future work.

Chapter 3

Single Receiver Design

The two primary objectives of Homa are low latency for short messages and high utilization of the network bandwidth when the offered load to the network is high. In the context of Homa, low latency means that tail latency for short messages (i.e. a few hundred bytes or less) is as close as possible to the latency that the hardware can offer in a unloaded datacenter network. In addition, the implications of the high bandwidth utilization goal is that large messages get the bandwidth they require to finish in a timely manner; even though Homa always tries to deliver short messages as quickly as the networking hardware allows, it does this quick delivery in a way that completion of large messages is minimally affected. To achieve the low latency goal, Homa's transport mechanism must act very efficiently in transmitting short messages while not wasting the network bandwidth that larger messages are striving for.

In this dissertation, we present the design of Homa in two phases. In the first phase, which is the focus of this chapter, we discuss how Homa achieves low latency through its receiver-driven packet scheduling and priority allocation mechanism. In the second phase of the design, which is the topic of Chapter 5, we *extend and complete* the design of this chapter, to support our second goal of high bandwidth utilization.

In order to focus on the low latency goal, the design in this chapter is restricted to a simplified many-to-one traffic pattern. In this simplified traffic scenario, there is only a single receiver in the network, and many senders are transmitting to this receiver. Later in Chapter 5, we extend this design to support a generalized many-to-many traffic pattern where we have many senders and many receivers in the network simultaneously. In the later chapter, we will explain how the over-commitment mechanism enables us to avoid wasted bandwidth and achieve high throughput goal in a network with arbitrary traffic patterns.

We emphasize that the restricted design in this chapter should not be taken as the complete and final design of Homa. This design helps us to explain many of Homa features and properties, but more detail and features will be added to this design in the later chapters.

3.1 Congestion: Where Does It Happen In Networks?

Congestion can theoretically happen at any output port of any switch in the network. Figure 3.1(a) shows an example of a two tier full-bisection leaf-spine datacenter network topology. Categorically, we divide the switch output ports in this network into two groups: 1) TOR (top-of-rack) downlink ports: the ports in the leaf switches at the link from the switch down to the individual host machines. 2) Core ports: any other output port in the switches which includes ports at uplinks from leaf to spine switches and downlinks from spine to leaf switches in the figure. In a datacenter network, congestion can theoretically happen in both TOR downlinks or at the output ports in the core of the network.

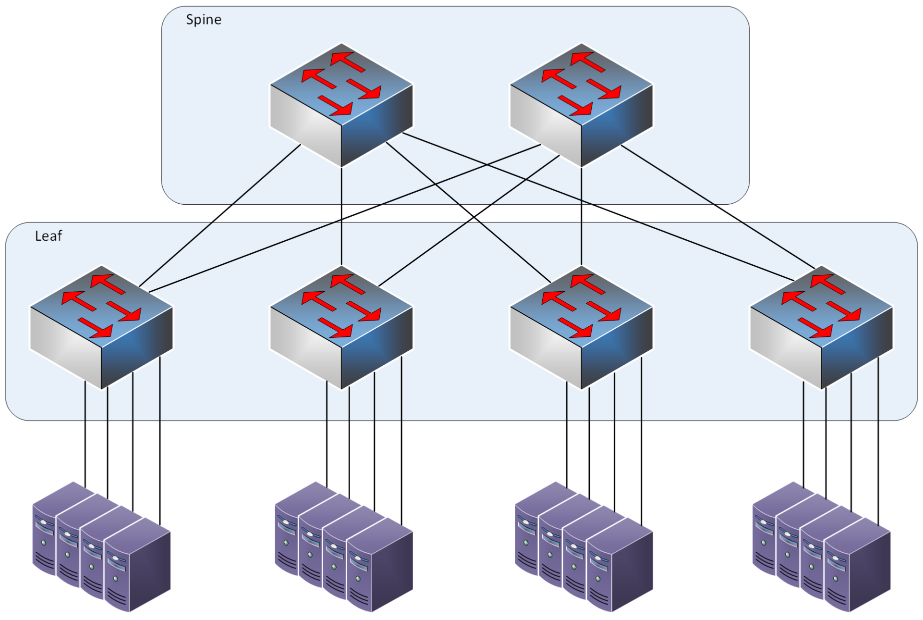
In datacenter networks congestion primarily happens at the TOR downlinks to the individual host machines; i.e. the first category of the ports in Figure 3.1(a). When multiple sender hosts simultaneously transmit messages at high rate to the same receiver host, they can easily oversubscribe the TOR downlink to the receiver. This overload on the link leads to packet build up at the top of the rack (TOR) queue near the receiver. Therefore the primary point of buffer congestion in the network is at the TOR downlink queues to the receivers. Homa's transport mechanism needs to manage this TOR buffer build up to avoid persistent congestion at the downlink. We note that this congestion can happen regardless of how much aggregated bandwidth the core has or how load balanced the core is.

We postulate that congestion is rare in the second group of the ports; i.e. the core of the datacenter networks. This is particularly a valid assumption if the following two conditions hold true in the network fabric: 1) The core of the network has packet level load balancing (e.g. packet spraying). 2) The core of the network has sufficient bandwidth to accommodate the offered load. As a result of these two conditions, we can safely claim that there will be *no persistent congestion at the core* of the network. To help understand why this claim is valid and the conditions are practical, we provide an example scenario that satisfies the two conditions. In the example of Figure 3.1(a), to route each packet from a source host to a destination host, the packet has to first travel upward toward the spine switches and then travel downward to the destination host. If at any hop on the packet's path toward the spine switches, we route the packet on a random uplink, the load balancing condition is satisfied. Moreover, because the core has a full bisection bandwidth fabric, none of the links in the network's core can become persistently oversubscribed with more load than it can transmit. Therefore, persistent congestion never happens in the core.

One caveat with these assumptions is that most datacenter operators don't build full bisection bandwidth networks because of financial considerations and cost limitations. They typically create oversubscribed networks with oversubscription ratios between $\frac{1}{2}$ and $\frac{1}{3}$. This ratio is computed as the fraction of aggregated bandwidth between leaf and spine switches, over aggregated bandwidth between leaf switches and host machines. In a full bisection topology like in Figure 3.1(a), this ratio is 1.

We note that a full-bisection bandwidth network is not a requirement for a congestion free network core; even in presence of an oversubscribed network, there are many ways to ensure that the core is congestion free. For example, in practice in datacenter networks, the host links typically run at less than 50% bandwidth utilization. Furthermore, some traffic is local within the rack and doesn't traverse through the network core at all. Therefore, we claim that an oversubscription ratio of $1/2$ is sufficient to avoid persistent congestion in the core. This claim holds true if we have optimal packet level load balancing available in the core. A large body of prior works has focused on the problem of fine-grained and efficient load balancing in the core of the network [11, 17, 2]. Utilizing ideas from these prior works in this domain, should enable us to fulfill a congestion free network core.

While fine-grained load balancing can be problematic for traditional transports (e.g. TCP), we designed Homa to be invulnerable to it and to take advantage of it. With traditional transports, fine-grained load balancing can cause difficulties in recovering from packet drops which increases latency. Let's explain why: an important job of a transports is to detect packet drops in the network as quick as possible and retransmit them; failure to retransmit them quickly, increases latency of the messages that experience packet drops. Meanwhile, traditional transports in order to detect packet drops, heavily rely on in-order delivery of the packets within a flow by the network. However, fine-grained load balancing schemes, like packet spraying, cause a lot of out of order packet deliveries within flows which interferes with packet drop detection and as a result increases message latencies. To mitigate these issues, we designed Homa to be invulnerable to out-of-order packet delivery by using reordering message buffers. Chapter7 elaborates on the details of this design and how Homa handles out-of-order packet deliveries.



(a) A schematic of a small but practical datacenter network topology that connects 16 host machines. The topology in this example is a two tier leaf-spine with edge links running at 10Gbps speed and core links running at 20 Gbps speed.

Host's TX Delay	Link Propagation Delay	Switching Delay	Host's RX Delay
1.0μs	0.1μs	0.25μs	0.5μs

(b) Fixed delays that a packet encounters while traversing the network of figure 3.1(a). The delays are measured from real packet traces in our experiments with RAMCloud [27] storage system and we use them as delay parameters in network simulations of Homa. "Host's TX delay" models the minimum delay a packet experiences in the host's software and hardware, from when it's sent in the transport until NIC starts transmitting it on the link. "Host's RX delay" models the minimum delay for a packet from when it's received in the NIC until it's delivered to the transport. With these delays, RTT in the network is computed to 8.2μs. RTT is computed as the delay of a full size 1538-byte Ethernet packet in one direction, plus an 84-byte acknowledgment packet in the other direction.

Figure 3.1

3.2 Homa's Congestion Control

To minimize buffer build up at the TOR downlink queues, Homa uses a receiver-driven congestion control mechanism; i.e. most of the congestion control logic resides in the receiver. The goal of Homa's congestion control is to manage buffer build up at the TOR downlink queue to the receiver host, when multiple sender hosts are transmitting packets to the receiver host. To realize this objective, a receiver-driven approach makes perfect sense because: among the receiver end-hosts and the sender end-hosts (the only two types of end-hosts in the network), receivers are the closest ones

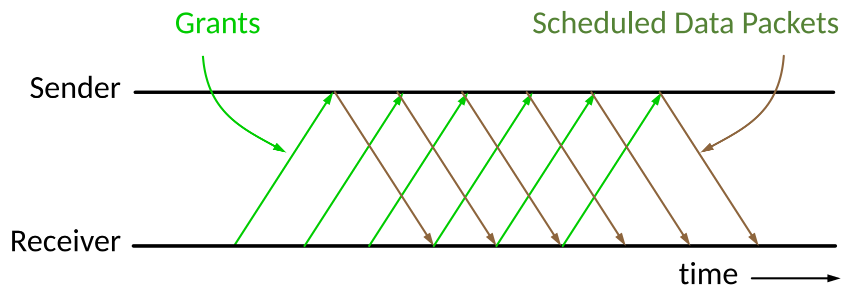


Figure 3.2: The Homa scheduler for a scenario with one sender and one receiver. The receiver paces the traffic from the sender by sending explicit grant packets. The sender sends a scheduled data packet for every grant packet it receives. The receiver sends a new grant packet every time it receives a data packet from the sender.

to the points of congestion at the TOR downlink queues. Therefore, they can have the appropriate context and information to control the congestion. This is in contrast to the TCP congestion control mechanism that is mainly driven by the sender hosts.

Before explaining how the receiver scheduler controls the congestion, let's take a step back and explain how exactly the scheduler behaves in the next subsection. Later in this section, we'll explain how packet scheduling allows Homa to control the congestion.

3.2.1 Receiver Driven Packet Scheduling

The behavior of a receiver scheduler is as follows: Each receiver host schedules the incoming traffic on its downlink, on a packet by packet basis. This behavior is illustrated in Figure 3.2 for a scenario with one sender host and one receiver host. As shown in the figure, for every data packet that the sender wants to transmit, the receiver explicitly sends a grant packet that allows transmission of the data packet. The data packets that are transmitted in response to grant packets, are called "scheduled data packets". Each grant packet specifies the number of data bytes the sender is allowed to encapsulate in a scheduled data packet. The receiver sends a new grant packet every time a new data packet is received from the sender. This allows the receiver to space the grants in time so that the incoming traffic rate matches the speed of the TOR's downlink to the receiver.

The receiver schedules the data packets as a part of a message. As we explained in Chapter 2, a message is defined as a block of data bytes that an application hands over to the transport in its entirety for transmission and the size of the block is known prior to transmission. For example in an RPC, the request part and the corresponding response part, each is a message.

The receiver scheduler has a bootstrapping problem: how does the receiver know when to start

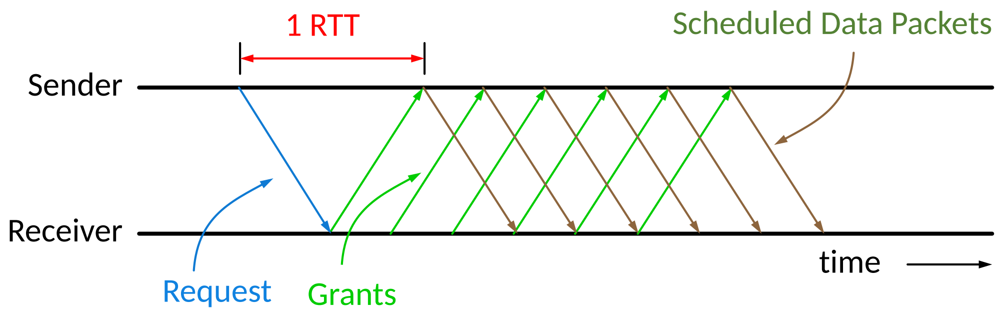


Figure 3.3: How does the receiver know how many grant packets it should send to the sender and when to start sending these grants? One approach is that each sender sends one Request packet and notifies the receiver. This approach, unfortunately, adds one RTT to the latency for the message.

sending grant packets to the sender and how many grants it must send? The receiver has to know that the sender wants to transmit a message, and it has to know the size of that message. One approach to communicate this information with the receiver is to have the sender notify the receiver by sending a tiny request packet. That packet includes the number data bytes in the message. The receiver then starts sending grant packets after it receives this request packet. Figure 3.3 shows this approach for solving the bootstrapping problem.

The problem with the approach of Figure 3.3 is that it adds one round trip time (RTT) of latency for every single message; it takes one round trip time from when the sender sends the request packet until it can send the first data packet (i.e. until the first grant packet from the receiver arrives at the sender). This latency overhead is particularly unacceptable for short messages. For example, a 100-byte message finishes with a latency of 3RTT/2 under this approach. However this message could have been transmitted with a latency of RTT/2 if it was blindly transmitted without any wait for grant packets. In this particular case, the scheduling approach has tripled the latency of the message, compared to the minimum possible latency for that message.

To avoid the latency overhead from scheduling, Homa allows each sender to blindly transmit a few data packets for each of its messages. These packets are called "unscheduled data packets" because the sender doesn't need to wait for permission from the receiver (i.e. grant packets) to transmit these data packets. Each sender, for each of its messages, is allowed to send enough unscheduled data packets to cover the RTT latency overhead depicted in Figure 3.3. These unscheduled packets also carry the size of the message, which informs the receiver about how many grants it needs to send to the sender. Figure 3.4 shows how sending the unscheduled packets allows Homa to avoid the one RTT latency overhead from scheduling and achieve the lowest possible latency for the sender's message. As shown in Figure 3.4, the number of data bytes in unscheduled packets is equal to RTTBytes. RTTBytes is defined as $RTT \times r$ where r is the host link speed in the network and RTT is computed as the sum of the transmission delays of a full-size 1538-byte Ethernet packet in one direction and an 84-byte acknowledgment packet in the other direction.

3.2.2 Receiver Side Congestion Control

With the restricted many-to-one traffic pattern of this chapter, Homa's receiver-driven scheduler can be leveraged to manage buffer build up at the TOR's downlink, by scheduling a single sender at time. As we discussed earlier in this chapter, when multiple distinct senders simultaneously transmit packets to the same receiver, they can oversubscribe the TOR's downlink to the receiver and build up congestion at the downlink. In Homa, the receiver has no control over unscheduled packets and they can create congestion when multiple senders simultaneously transmit them. However, the receiver has total control over the scheduled packets and it can prevent the congestion from further blowing up, by sending grants to a single sender at a time; the receiver can pace its inbound scheduled traffic rate and ensures that its downlink does not remain oversubscribed from arrival of scheduled data packets from multiple senders. So, each sender sends RTTBytes of unscheduled data packets and then waits for grant packets from the receiver. The receiver sends a grant to a single sender, whenever a data packet arrives from any of the senders. The net effect is that the receiver prevents persistent growth of the queue size at its downlink and controls the congestion; congestion is capped at RTTBytes times the number of senders (i.e. total unscheduled bytes from the senders). We call this type of scheduling, "monogamous scheduling" where the receiver only schedules a single sender at a time.

3.3 Receivers Use SRPT Scheduling

With the monogamous scheduling, each receiver has a decision to make: among all senders waiting for grants, which sender should be granted first? The scheduling order in Homa is determined based on SRPT policy. SRPT stands for Shortest Remaining Processing Time first; it refers to a type of policy that focuses on the job with the shortest remaining time to complete and finishes that job first, before any other job (in the context of network transports, transmission of each message from its sender to its receiver is a job). SRPT schedulers are a subtype of preemptive schedulers. Specifically, if an SRPT scheduler is currently serving a job J_1 that has the shortest remaining size to complete and a new job J_2 arrives at the scheduler that is even shorter than the remaining size of J_1, the

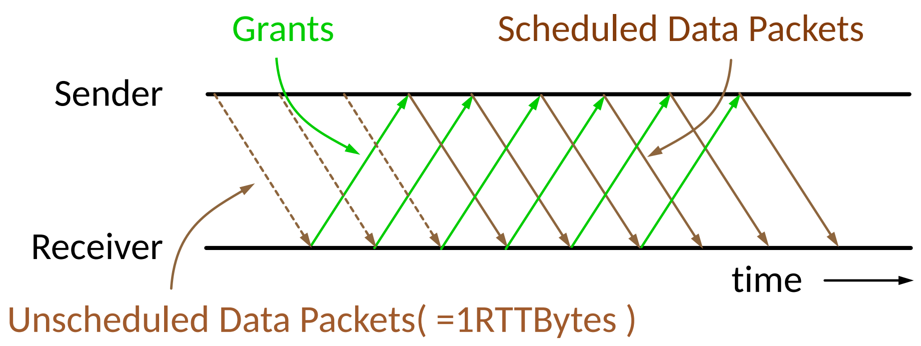


Figure 3.4: To avoid the latency overhead of receiver-driven scheduling, each sender is allowed to transmit RTTBytes of unscheduled data packets for each of its messages. This allows the messages to finish in minimum possible latency in an unloaded network. As an example, in a datacenter network where RTT is $10\mu s$ and host links run at $10Gbps$, RTTBytes is 12500Bytes.

scheduler stops serving (or preempts) J_1 in favor of J_2 and starts serving J_2.

An important property of SRPT is that it leads to very low latency for short messages, thus Homa adopts SRPT as its scheduling policy. The example in Figure 3.5-a illustrates how a Homa receiver scheduler implements SRPT to favor scheduling the shorter message, when two messages from different senders are waiting for grants. In this figure, initially the receiver is only receiving packets from a large message from Sender 1 and therefore it sends a new grant packet for that message, each time it receives a data packet from that sender. But, at time T_4 the first packet from Sender 2 arrives at the receiver and the receiver now has a decision to make: among the two senders waiting for grant packets, in what order should the receiver scheduler send grants to them? Since low latency for short messages is one our most important goals, the receiver scheduler tries to dedicate its bandwidth to the shorter message. To that end, the receiver stops sending grant packets to the long message from Sender 1 (i.e. preempts it) and starts sending grant packets to shorter message from Sender 2. One the shorter message has been fully granted, the receiver resumes granting the larger message from sender 1.

Homa's scheduling policy as depicted in Figure 3.5-a can be generalized to more than two senders. When multiple messages from different senders are waiting for grant packets, Homa chooses the message with the shortest remaining bytes to receive and schedules that message first; every time a grant packet can be sent, the receiver sends the grant to the message that is nearest to completion. If the receiver has been sending grant packets to a message and all of sudden another message arrives with a shorter remaining bytes to receive, the scheduler preempts the larger one in favor of the shorter message, and starts sending grant packets to the shorter message.

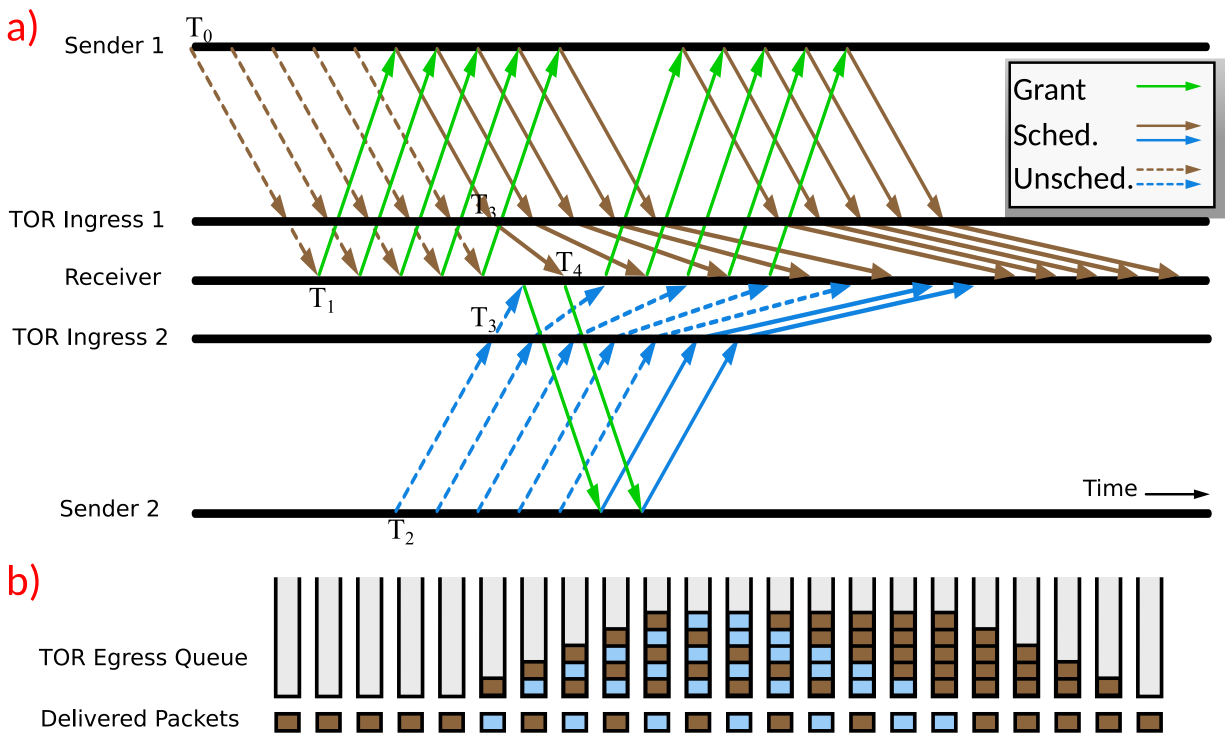


Figure 3.5: Congestion control and SRPT scheduling when two senders are transmitting packets to one receiver. Time is increasing from left to right and the timelines of Sender 1, Sender 2, and the Receiver are shown. The other two timelines represent the ingress ports of the Receiver's TOR on which packets from sender 1 and sender 2 arrive. RTTBytes in this hypothetical network is 5 packets. TOR queues are FIFO, no priority queue is used.
a) In this scenario, Sender 1 starts transmitting unscheduled packets at T_0. The Receiver begins granting Sender 1 at T_1. Sender 2 starts sending unscheduled packets at T_2 and packets of both Sender 1 and Sender 2 arrive at the TOR at T_3. The receiver preempts Sender 1 and starts granting the shorter message from Sender 2 until it's fully granted at T_4. After this time, the Receiver resumes granting Sender 1 again.
b) Preemption lag occurs if packets of a shorter message are delayed by packets of larger messages in the queue. Packets from senders pass through the TOR queues, and are transmitted to the receiver at the bottom. The interleaving packets from Sender 1 increase the latency of Sender 2's message by one RTT. The bins show how the TOR queue length grows in time, as new packets arrive at the TOR's ingress ports. The queue starts to build up at T_3 when the packets from both Sender 1 and Sender 2 arrive at the TOR ingress ports. The queue size grows to as large as RTTBytes, from a mixture of packets from both senders. RTTBytes of packets from Sender 1 are delivered to the Receiver, interleaving the packets from Sender 2. This increase Sender 2's message latency by one RTT.

One of the desirable properties of SRPT schedulers is “run to completion” behavior. Run to completion means that the scheduler favors the message that has transmitted more bytes than others. Therefore, run to completion leads to low latency for the messages that keep making the most progress. This behavior is automatically realized with Homa’s SRPT scheduler. Run to completion behavior is not fair for transmission of messages, but it yields to a better end-to-end system design and performance. In contrast to round-robin schedulers that lead to fairness among messages, SRPT schedulers lead to run to completion behavior which improves the latency of messages. The importance of run to completion in improving latencies can be better understood when we’re scheduling messages of equal sizes; once the SRPT scheduler chooses one of the messages of equal sizes and schedules one packet from it, the scheduler consequently favors that message and dedicates its bandwidth to completely transmit it. Not only this minimizes the latency of the message that sent more bytes, but also compared to a fair scheduler, it reduces mean, median, and tail latencies among those messages (with a fair scheduler all of the messages would have completed slowly).

The grant mechanism of Homa’s receiver schedulers allows them to preempt the larger messages for the shorter messages. Hence, we call this type of preemption, ”preemption by grants”. Although ”preemption by grants” allows Homa receivers to achieve lower latency for shorter messages, unfortunately it does not lead to the smallest possible latency for short messages. This is a problem that we discuss in the next section and we’ll resolve it by utilizing network priority queues.

We should note that the monogamous scheduler is not the final design of Homa’s scheduler. We’ll show later in simulations that this scheduler accurately controls the congestion and closely implements SRPT under the many-to-one traffic pattern of this chapter. But as we discussed earlier, the main benefit of the monogamous scheduler and many-to-one traffic patterns is that they allow us to pay our undivided attention to the design and analysis of Homa’s priority assignment mechanism which is the topic of the rest of this chapter. That said, this scheduler has other issues that would surface up under many-to-many traffic patterns and we need to address them. In Chapter 5, we will extend Homa’s scheduler with the concept of controlled overcommitment to fix those issues. We will demonstrate that with the controlled overcommitment mechanism, Homa is still able to manage the congestion and implement SRPT, but the queue sizes increase in the TORs.

3.4 Priorities For Preemption

The ”preemption by grants” mechanism has a problem that we call “preemption lag” which increases the latency of shorter messages. “Preemption lag” means that preemption does not happen immediately; a few packets from the larger message, which were already in flight, arrive at the receiver before those from the shorter message. In other words, the receiver scheduler cannot fully preempt all packets of the larger message, when it stops sending grant packets to it and starts granting the shorter message. ”Preemption lag” happens as a result of temporary buffer build up at the FIFO queue of TOR downlink to the receiver.

Figure 3.5-b illustrates why preemption lag occurs in our example of two senders and one receiver. The packets in the bins in this figure show how the TOR’s queue length changes as new packets arrive at the TOR. In the figure, when the receiver preempts Sender 1 at T_3 and starts granting Sender 2, there are already RTTBytes in packets in-flight from Sender 1 that were previously scheduled by the receiver. These packets from Sender 1 arrive at the TOR simultaneously with the packets from Sender 2. The simultaneous arrival of packets from the two senders results in a short-lived over-subscription of TOR’s downlink for one RTT time. This oversubscription in turn causes a small queue of size RTTBytes to build up at the TOR’s downlink to the receiver. This means RTTBytes worth of packets from Sender 1 are interfering with the packets from Sender 2 and the TOR’s FIFO queue delivers the Sender 1’s packets among the packets from Sender 2, according to the order they arrive at the TOR ingress ports. Hence, completion of sender 2’s message is delayed by one RTT.

When we design for low latency, transmission of packets from shorter messages must always be prioritized over transmission of packets from larger messages. So the transport should be designed in a way that prevents packets of large messages from blocking packets of shorter messages. To motivate why this is a necessary design decision, let’s consider a practical scenario in the network of Figure 3.1(a). Table 3.1(b) shows a breakdown of different fixed delays in the network (these delays are measured from real packet traces in our experiments with RAMCloud storage system). According to these delays, a short 100-byte message can be delivered in less than $1.7\mu s$ on the longest path in the network that passes through three switches. However, serialization of a full-size 1500-byte packet can take $1.23\mu s$, about 72% of the short message latency; if the short message is blocked by one full packet, even at one hop in the network, the latency of the shorter packet increases by 72%. At tail, the latency of the short message could be even worse since it could be blocked multiple times by full packets on different hops in the network. This is a form of preemption lag at packet level granularity which causes the latency of short messages to increase. One the key ideas we pursue in Homa’s design is to prevent preemption lag at packet level granularity, in order to achieve low latency for short messages.

Fortunately commodity network switches, through network priority queues, provide the means by which we can eliminate preemption lag. Network physical layers such as Ethernet, have long standardized these priority queues (in Ethernet they're also referred to as Quality of Service or QoS levels). For instance, in Ethernet switches and routers, each egress port of the switch/router contains eight priority queues, one queue per priority level. Every Ethernet packet, prior to transmission at the sender transport, can be tagged with a priority level. When the packet is transmitted and the switch receives it, the switch arbiter inspects the priority level of the packet, then forwards it to the appropriate egress port and buffers the packet in the queue that matches the priority tag of the packet (refer to IEEE 802.1Q for the details on priority levels in Ethernet Layer). Whenever the switch wants to send a packet out at that output port, it always strictly favors packets in the higher priority queues to lower priority queues. Throughout this dissertation, we assume at most eight priority queues P_1 to P_8 per switch port, and we always have P_1 as the lowest priority queue and P_8 as the highest priority queue (switch always forwards packets from P_8 before any other queue; if P_8 is empty, then it forwards from P_7 and so on).

Figure 3.6 shows how utilizing priorities can help us to achieve packet level preemption of larger messages and optimum latency for shorter messages. In this example, suppose that in order to allow fast and unblocked delivery of unscheduled packets from Sender 2's message (i.e. the shorter message), Sender 2 knew that it had to tag its unscheduled packets with a priority level higher than Sender 1's packets. Therefore, as the packets from the two senders arrive at the TOR buffer, Sender 2's unscheduled packets preempt the packets of Sender 1. A queue of size RTTBytes forms at the TOR downlink buffer to the receiver, however, that buffer is strictly from the Sender 1's lower priority packets. After the Receiver receives the first packet of Sender 2, it preempts Sender 1 (i.e. stops sending grant packets to it) and starts sending grant packets to Sender 2. In order to ensure that scheduled packets of Sender 2 are not blocked by the packets of the Sender 1 message, the scheduled packets of Sender 2 must also be higher priority than the scheduled packets of Sender 1. In such case, the scheduled packets of Sender 2 also preempt the scheduled packets from Sender 1 at the TOR queue and the entire message of Sender 2 can be delivered to the Receiver without any interruption. Hence, the packet level preemption using priority queues and QoS levels allows us to achieve optimum latency for the shorter message. That said, the question remaining to be answered is how does Sender 1 knows what priority level it should use, that is the topic of the next section.

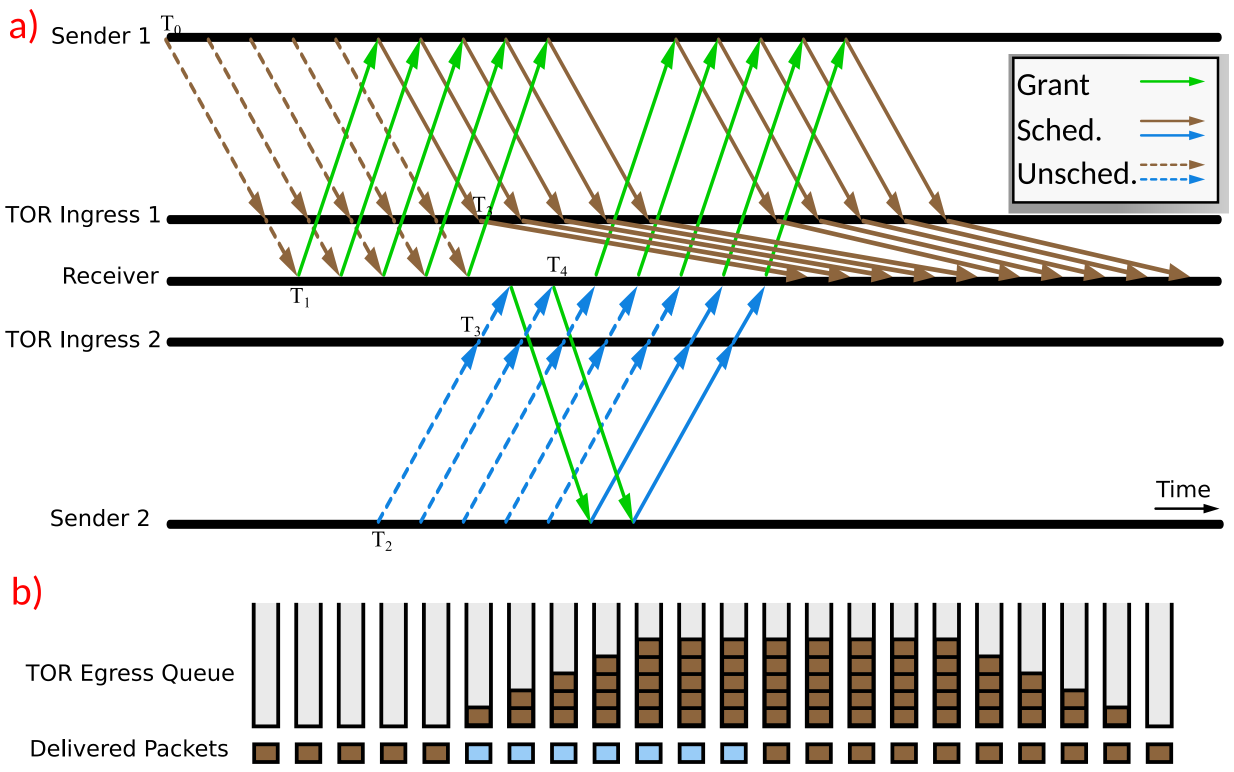


Figure 3.6: Congestion control when two senders are transmitting packets to one receiver. Time is increasing from left to right and the timelines of Sender 1, Sender 2, and the Receiver are shown. The two other timelines represent the ingress ports of the Receiver's TOR on which packets from sender 1 and sender 2 arrive. RTTBytes in this hypothetical network is 5 packets.
a) In this scenario, Sender 1 starts transmitting unscheduled packets at T_0. The receiver starts sending grants to Sender 1 at T_1. Sender 2 starts sending unscheduled packets at T_2 and the packets of both senders arrive at TOR at T_3. Since the unscheduled packets of Sender 2 are tagged with a higher priority level, they preempt Sender 1's packets. The receiver preempts Sender 1 and starts granting the shorter message from Sender 2 until it's fully granted at T_4. Since the packets of Sender 2 are all tagged with a higher priority level than Sender 1's packets, they all arrive back to back without any interruption from Sender 1's packets. The receiver resumes granting Sender 1's message again when Sender 2 is fully granted.
b) Shows how the packet queue builds up in time up at the TOR's buffer near the receiver. The queue starts to build at T_3 when the packets from both Sender 1 and Sender 2 arrive at the TOR buffer. Higher priority packets from Sender 2 preempt Sender 1's packets, so the queue forms solely from packets of Sender 1. Higher priority for Sender 2's packets allow them to immediately get delivered to the receiver as soon as they arrive at the TOR; the net effect is optimal latency for the short message from Sender 2.

3.5 Priority Assignment Mechanism

The example in Figure 3.6 is a simple scenario that demonstrates how utilizing network priority queues can lead to perfect preemption in packet level granularity and low latency for the shorter message. However, the question that remains to be answered is: How can we generalize this example to a more concrete and crisp mechanism for allocating priority levels among packets of various message sizes?

Network priority queues are the corner stone of Homa's transport mechanism; they are the most novel feature in Homa, and the key to its performance. But before jumping into the details of how we use them, we need to discuss several limitations and considerations about them. Below we present at high level, some of these limitations and consideration of priority queues and we explain in what ways we should use them in Homa.

Priorities Are Limited

Homa is designed to be a practical transport scheme for datacenters and it must perform within the limitations of the network fabric. That means, Homa should only rely on the features and capabilities provided by commodity network hardware. Priority queues have been a feature of the commodity network hardware for many years but with one caveat: the number of priority levels is limited. For example, as we discussed earlier Ethernet switches and NICs only standardize and provide eight priority levels. Therefore, priorities are precious and scarce resources and Homa should be designed to work efficiently with a limited number of priority levels.

Priorities To Implement SRPT

As we discussed in the previous section, Homa tries to approximate the SRPT scheduling policy to achieve low latency for short messages. Homa's priority assignment scheme is also designed to get a closer approximation of the SRPT policy. This means, as the message gets closer to completion, the remaining packets should receive a higher priority than earlier packets.

Receivers Allocate Priorities

As we discussed earlier, the primary point of congestion in the network is TOR downlinks to receivers and Homa's goal is to implement SRPT by assuring that packets of short messages can preempt packets from larger ones at the congested queues. This implies that the receivers are the right place to assign priority levels for packets; each receiver knows which senders are transmitting to it and it also knows the sizes of the messages from the senders. Each receiver determines the priorities for all of its incoming data packets in order to approximate the SRPT policy; the receiver allocates higher priorities for packets from shorter messages.

Priorities For Both Scheduled and Unscheduled Packets

As we discussed earlier, Homa's data traffic is divided into two categories of unscheduled and scheduled packets, and priorities must be used for both of these packet types. To find a suitable priority allocation for each of the packet types, Homa' priority assignment scheme needs to address the two issues below:

1. Receivers need to use separate priority allocation schemes for unscheduled and scheduled packets. That's because unscheduled packets are transmitted blindly without the receiver's prior knowledge and receiver has no instant control to set priorities of these packets. Whereas, the scheduled packets are transmitted with the permission of the receiver and the receiver has full control over the priority level that should be used for each scheduled packet.

2. Because the number of priority levels are limited and Homa has separate priority allocation mechanisms for scheduled and unscheduled packets, then Homa needs to decide how many of the priorities are used for unscheduled packets and how many of them are used for scheduled packets.

In the rest of this section, we present Homa's priority allocation schemes and discuss how they address the above limitations and considerations.

3.5.1 Priority Allocation: Scheduled vs. Unscheduled

The first question that we need to answer is how should the receiver divide the priority levels between scheduled and unscheduled packet? More specifically, the receiver needs to decide that:

1. From a limited number of priority levels, how many of them need to be used for unscheduled packets and how many of them need to be used for scheduled packets?

2. Should unscheduled packets have strictly higher priority than scheduled packets? Or, should scheduled packets have strictly higher priority than unscheduled packets? Or, should a mixture of higher and lower priority levels to be used for both scheduled packets and unscheduled packets?

Homa allocates the higher priority levels for unscheduled packets. For example, suppose that P_1, P_2, ..., P_8 are the total available priority levels, with P_8 being the highest and P_1 being the lowest priority level. Homa chooses to allocate P_1, ..., P_k for scheduled packets and P_{k+1}, ..., P_8 for unscheduled traffic, with $1 < k < 8$. k is the index of the highest scheduled priority and later in this section we will explain how it is computed.

There are two reasons for why Homa chooses higher priority levels for unscheduled traffic. The first and primary reason is the fact that datacenter applications have lots of short messages that can be fully transmitted in unscheduled packets and we expect to transmit them at the lowest possible latency. When we study practical message size distributions in the datacenter networks, we find that many of these workloads have lots and lots of messages with sizes that are equal or smaller than RTTBytes. For example, RTTBytes (i.e. unscheduled limit for each message) in the network topology of Figure 3.1(a) is equal to 10 kbytes and when we inspect the workload distributions of Figure 2.1(a), we see that in W1 to W6 (6 out of 7 workloads), messages smaller than 10 kbytes are at least 62% of all messages. Allocating higher priority levels for unscheduled packets allows this high volume of short messages to preempt larger ones in the network and finish quickly with low latency.

The second reason for having unscheduled packets at higher priority is to let receivers know about the incoming messages as quickly as possible. Remember that unscheduled packets play the important role of letting the receiver know about a message and its bytes size. When the first unscheduled packet arrives at the receiver, the receiver inserts this message into its list of the outstanding messages that require grant packets from the receiver. The sooner the unscheduled packets are delivered, the better the receiver can make a decision for granting them. Allocating higher priority levels for unscheduled packets allows these packets to get delivered with low latency through the network. Therefore the receiver can make more informed scheduling decisions. Furthermore by having all of them at high priority, we better protect against unscheduled packet drops; even if all except one is dropped in the network, the receiver gets information gets required information for scheduling that message. In the evaluation Chapter 4 we will return to this design decision and inspect it in more detail.

The next question is how should a Homa receiver choose the number of priority levels for scheduled versus unscheduled packets (i.e. k, the index of highest scheduled priority)? Homa divides priority levels between unscheduled and scheduled packets to balance bytes proportionally between the two priority types. Each Homa receiver measures the incoming traffic on its downlink and computes the fraction of all incoming bytes that are unscheduled (i.e. the fraction of bytes delivered only in unscheduled packets over all delivered bytes in both scheduled and unscheduled packets). It then divides priority levels between scheduled and unscheduled packets according to this fraction; it allocates this fraction of the available priorities (the highest ones) for unscheduled packets, and reserves the remaining (lower) priority levels for scheduled packets. For example if a receiver measures that 80% of bytes are received in unscheduled packets and the remaining 20% in scheduled packets, it then uses 80% of higher priority levels for unscheduled packets and 20% lower priorities for scheduled priorities (rounded off to the nearest integer). In this example, if there are 8 priority levels, then the receiver allocates P_1, P_2 for scheduled packets and the remaining priority levels for unscheduled packets. If this scheme results in zero priorities for scheduled packets (or unscheduled packets), then we override this allocation scheme and assign one priority level for scheduled packets (or unscheduled packets). This allocation scheme is purely an empirical design that we will evaluate in more detail in Chapter 3, and we'll compare it against a few other possible designs.

3.5.2 Priority Assignment For Unscheduled Packets

Each Homa receiver allocates priorities in advance for unscheduled packets. It uses recent traffic patterns to choose priority allocations, and it disseminates that information to senders by piggybacking it on other packets. Each sender retains the most recent allocations for each receiver (a few dozen bytes per receiver) and uses that information when transmitting unscheduled packets. If the receiver's incoming traffic changes, it disseminates new priority allocations the next time it communicates with each sender.

To get a close approximation of the SRPT policy, the priority level for unscheduled packets of a message should increase as the remaining size of the messages decreases. A Homa sender needs to elevate the priority level assigned to unscheduled packets of a message as the message sends more and more unscheduled packets and the message gets closer to completion. For example, imagine two messages from two senders that each consist of five unscheduled packets. When Sender 1 transmits the first unscheduled packet, the remaining size of the message from the sender's perspective is four packets and shorter than the five packets message from the other sender. Therefore, in order to realize the run to completion behavior for Sender 1's message, Sender 1 needs to send its second unscheduled packet at a higher priority level to ensure this packet will not be blocked by the unscheduled packets from Sender 2. However, the number of priority levels are limited and we can't always guarantee a higher priority as we send more unscheduled packet. Therefore, the challenge is to assign priorities for unscheduled packets such that as a message sends more packets and gets closer to completion, it has a chance to get a higher priority level for its unscheduled packets, subject to the constraint that priorities are limited.

To assign unscheduled priority levels, Homa divides messages into several groups and messages in the same group use the same priority level for their unscheduled packets. Messages are grouped based on their remaining sizes at the sender side. Groups are identified by consecutive ranges of message remaining sizes. The number of groups is equal to the number of priority levels for unscheduled packets (since each group is assigned a single priority level). Each message depending on its remaining size at the sender falls into a group. However, the message may switch group as it sends more unscheduled packets; as the message sends more unscheduled packets (i.e. the remaining size of the message decrease), it is moved to the next group that is associated with the higher priority level and it uses the new priority level for its remaining unscheduled packets. This allows Homa to approximate SRPT with limited number unscheduled priority levels.

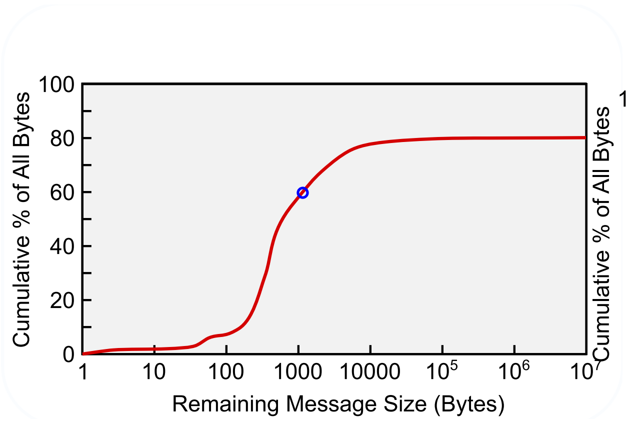
The interesting question to ask is how should we find the message groups? Homa computes message groups for unscheduled priority assignments so that each priority level is used for about the same number of transmitted bytes. As we discussed earlier, our goal is to compute consecutive ranges of remaining message sizes such that smaller ranges get higher priority level and we can approximate SRPT with limited priorities. This can be achieved by computing the CDF of the workload at the receivers. To determine the groups, each receiver records statistics about the sizes of its incoming messages and uses the CDF of workload message sizes to compute a transformation of the workload's CDF; Figure 3.7(a) shows an example of the transformed plot computed directly from a workload's CDF. The x-axis on this figure shows the remaining size of a message after sending a certain number of packets and the y-axis shows the cumulative percent of bytes in the network that are transmitted in the unscheduled packets. Using this plot, the receiver then chooses the groups so that each priority level is used for an equal number of unscheduled bytes and shorter messages use higher priorities. Figure 3.7(b) shows how groups are computed and unscheduled priorities are assigned to each group. In this example, we assumed that a total of six unscheduled priority levels are available: P_1 to P_6 with P_1 being the lowest priority level and P_6 the highest priority level. Homa chooses groups (i.e. ranges of message remaining sizes) on the x axis such the y axis is divided into equal ranges (i.e. equally spaced lines on the y axis equal amount of bytes per priority level). Messages with remaining sizes of larger than 1500 bytes are in the first group, and they send unscheduled packets at priority P_1 and messages with remaining sizes between 900 and 1500 bytes are the second group and they use priority level P_2 for their unscheduled packets and so on. In Homa, the group upper and lower bounds (e.g. 1500B, 900B, etc. in this example) are called priority cutoffs. This scheme of computing the groups and priority cutoffs guarantees that with limited priority levels, as a message sends more unscheduled bytes and the message's remaining size decreases, the message can fall into a new group with higher priority level and assign the higher priority level for its next unscheduled packets. Therefore, SRPT is approximated using limited number of priority levels.

Each receiver in Homa, periodically and independently from other receivers, computes the priority cutoffs and communicates them to its senders. The receiver collects statistics of the message sizes transmitted to it and computes the CDF of message sizes from the history of the sizes. From the computed CDF, the receiver can then find the plot of 3.7(a) and respectively the priority cutoffs as depicted in Figure 3.7(b). The receiver occasionally transmits these cutoffs to its active senders and the senders use these priority cutoff sizes for unscheduled packets they transmit to that receiver in the future. Since we do not expect the receiver workloads to change very frequently, the receiver periodically recomputes the priority cutoffs every tens of minutes and retransmits the updated priority cutoff sizes to its active senders. If a sender is new to the receiver or the receiver has not received a message from that sender for a long time, the receiver transmits the priority cutoff sizes to that sender, immediately after getting the first packet from it. The senders have to keep separate priority cutoffs for distinct receivers.

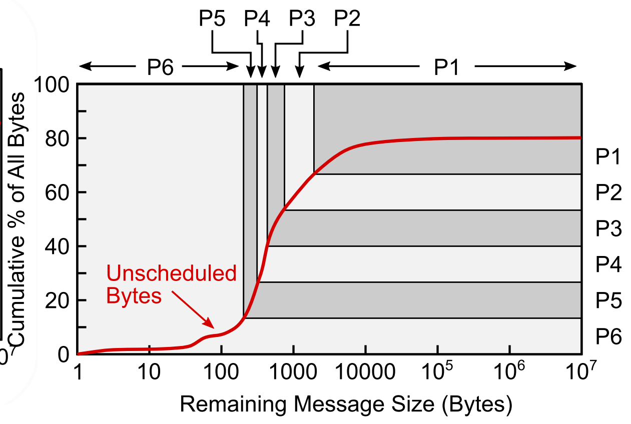
3.5.3 Priorities For Scheduled Traffic

In Homa, priorities for scheduled packets are assigned by receivers. Each receiver, independently from other receivers, sets the priority levels for its inbound scheduled packets. The receiver assigns the priority level for scheduled packets in each grant it sends for each packet. The sender of a scheduled packet, uses the priority level specified in the grant for the granted bytes.

A receiver dynamically adjusts the priority allocation, based on the precise set of messages being received, such that a higher priority level is used for scheduled packets of shorter messages. The receiver initially grants at the lowest scheduled priority level when it starts scheduling the first message. When it learns about a new message that has a shorter remaining size than the one currently being granted, it uses the next higher priority level in grants for the shorter message. This ensures that scheduled packets of the shorter message preempt the packets of the larger one (allows Homa to avoid preemption lag) and produces a better approximation of SRPT. When the higher priority message is fully granted, the receiver resumes granting the previously preempted larger message at the lower priority level. So the receiver can adapt to the changes in the inbound traffic using this



(a) The CDF of unscheduled bytes across messages of different remaining sizes for a given workload (i.e. W2 in Figure 2.1). 100% on the y-axis corresponds to all network traffic, both scheduled and unscheduled; about 80% of all bytes are unscheduled. The blue dot on the figure shows that 60% of total bytes in the network are transmitted in the unscheduled packets of messages that have a remaining size of 1000 bytes or less. This figure can be computed directly from the message size CDF.



(b) The unscheduled bytes CDF is used to determine the message groups (i.e. ranges of remaining message sizes or priority cutoffs) for each priority level so that traffic is evenly distributed between the priority levels (equally spaced lines on the y axis). For example, P6 (the highest priority level) will be used for unscheduled bytes for messages with remaining size 1-280 bytes and P1 (the lowest priority level) is for messages with remaining size larger than 1500 bytes.

Figure 3.7: Homa receivers allocate unscheduled priorities based on traffic patterns.

dynamic priority assignment based on the current set of ungranted messages, and it can achieve a closer implementation of the SRPT policy.

Figure 3.8 illustrates this mechanism. In this figure, the receiver time line is shown as the blue arrow. Over time the receiver receives packets from 4 different messages with sizes 200KB, 150KB, 50KB, and 100KB. The figure shows the packets of the messages as they arrive at the receiver. Each packet that arrives triggers a new grant to the highest priority message. The grant packets are shown as G_{Pi} where i is the scheduled priority level assigned by the receiver. In the figure, the first packet arrives at time T_1, from the first message with size 200KB. Immediately after delivery of the first packet, the receiver starts granting this message at the lowest scheduled priority possible P_1. At time T_2, the first packet from a 150KB message arrives. Since this message has a shorter remaining size than the previous message, the receiver grants this message at the next higher priority level P_2. This ensures a perfect preemption for this message as the scheduled packets of this message wont get blocked by the previous message. The same behavior happens at T_3 when the receiver learns about 50KB message that has a shorter remaining size than both of the previous messages. At this point the receiver uses the higher scheduled priority P_3 and grants the new message at this priority level. Therefore, the receiver can avoid the preemption lag every time a new high priority message

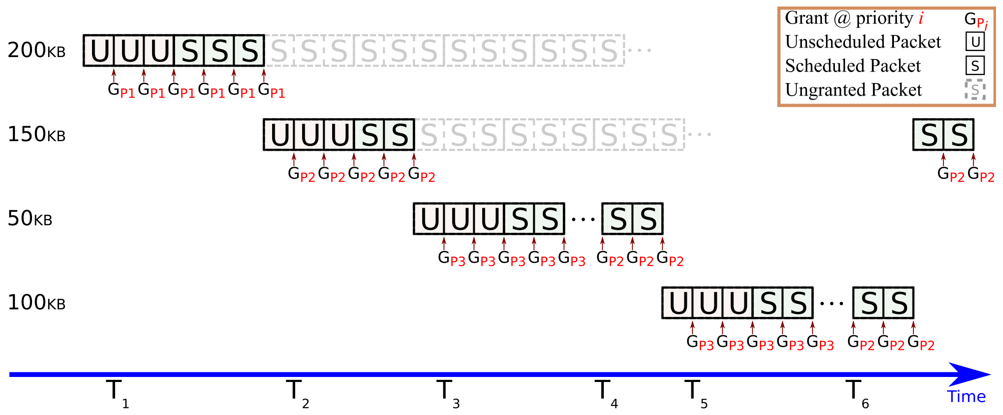


Figure 3.8: Receivers dynamically assign higher priorities for scheduled packets of shorter messages. The time line of a Homa receiver is depicted and the diagram illustrates the transmission of four messages to the receiver over time. The sizes of the messages are shown on the left y axis of the figure. Each message is packetized: U packets are unscheduled data packets and S packets are scheduled data packets. Whenever a data packet is received, it triggers transmission of a grant packet to the message with shortest remaining size; ↑ shows when the grant is sent. Each grant is subscribed by the priority for the scheduled packet (e.g. G_{P2} means a grant packet for scheduled packet at priority $P2$). When the first packet of a shorter message arrives, the receiver uses a higher priority level.

is presented to it. Once the 50KB messages is fully granted at time T_4, the receiver resumes granting the 150KB message at priority P_2 again.

Note that a receiver only needs to use a higher priority level if a message arrives that is shorter than all ungranted messages. It is an acceptable behavior for the receiver to schedule two messages with different sizes on the same priority level when the packets of the two messages can't interfere with each other. This behavior is shown in the figure 3.8 at time T_5, when the receiver learns about a message with size 100KB. The receiver grants this message at priority P_3 that was previously used for scheduled packets of the 50KB message. The receiver reuses P_3 without running the risk of preemption lag for the shorter 50KB messages because the shorter 50KB message is fully granted and completed when the 100KB message arrives. At that point the 100KB is the highest priority message among the messages that needs to be granted packets. This ensures that the packets of this 100KB message can preempt the packets of the 150KB and 200KB messages while the packets of 100KB can't interfere with the packets of 50KB message. Finally at T_6 when the 100KB is fully granted, the receiver starts granting 150KB message again at the same priority P_2 it was granted in that past.

If there are fewer messages than scheduled priority levels, then Homa uses the lowest of the available priorities; this leaves higher priority levels free for new higher priority messages. If Homa always used the highest scheduled priorities, it would result in preemption lag: when a new higher priority message arrived, its scheduled packets would be delayed by 1 RTT because of buffered packets from the previous high priority message (see Figure 3.5-b).

One issue with this dynamic scheme is that the number of preemptions is limited by the total number of available scheduled priority levels; when all scheduled priorities are in use (i.e. the number of ungranted messages is equal or larger than the number of the priorities), then two or more messages (the shortest ones) share the same priority level. This means the shortest message experiences preemption lag. For example, suppose that there are three total priority levels available for the scheduled packets. If four messages with sizes 250KB, 200KB, 150KB, and 100KB consecutively arrive at the receiver, the receiver can achieve perfect preemption among the first three messages by scheduling the 250KB message at P_1, the 200KB message at P_2, and the 150KB message at P_3. However, when the 100KB message arrives, the receiver has already exhausted all of the available priority levels. Therefore this message has to be scheduled at P_3 and share this priority level with RTTBytes packet of the 150KB message. The limited number of priority levels means limited number of perfect consecutive preemptions. This in turn means there is still possibility of preemption lag and less efficient approximation of the SRPT policy. We will study this limitation in more detail in evaluations of Chapter 4.

3.6 Senders Also Use SRPT Scheduling

Earlier in this chapter we described the behavior of senders. Here, we summarize that behavior again: when a message arrives at the sender's transport module, Homa divides the message into two parts: an initial unscheduled portion (the first RTTbytes bytes), followed by a scheduled portion. The sender transmits the unscheduled data bytes immediately in one or more packets. The scheduled data bytes are not transmitted until requested explicitly by the receiver using grant packets. Each data packet has a priority, which is determined by the receiver as described in previous sections.

To avoid head of line blocking for short messages at senders' uplinks, the senders also implement SRPT for their outgoing packets: if data packets from several messages are ready to be sent, packets for the message with the fewest remaining bytes are transmitted first. Control packets such as grants are always transmitted a head of all data packets. To implement SRPT, senders do not consider the priorities in the data packets when they transmit packets. That's because the priorities in the packets reflect the receivers' priorities (i.e. priorities in packets are intended to achieve SRPT for the final downlinks from TORs to the receivers); a sender's priority for an outgoing packet is different from to the receivers' priority for that packet.

For a sender to implement SRPT precisely, it must keep the transmit queue in the NIC as short as possible. That's because The NIC's transmit queue is FIFO (it does not use packet priorities) and if a queue builds up in the NIC, then the packets of short messages may have to wait for packets of larger messages that were queued previously. To keep the NIC's transmit queue short, Homa's transport mechanism has an unusual feature which limits buffer buildup in the NIC transmit queue. Homa keeps a running estimate of the total number of untransmitted bytes in the NIC, and it only hands off a packet to the NIC if the number of untransmitted bytes (including the new packet) is less than two full-size packets. This allows the sender to reorder outgoing packets when new messages arrive for transmission. In evaluations of Chapter 7, we show that if we disable this feature of Homa, short messages' tail latencies may increase by 100x.

We maintain a non-empty packet queue in NIC (i.e. less that two full-size packets) to avoid wasting bandwidth at the NIC's uplink. Ideally, we'd like to keep the NIC's transmit queue empty to avoid any head of line blocking; i.e. hand off a new packet to the NIC, exactly when the NIC has transmitted the last byte of the previous packet. But while we were designing Homa, we realized that keeping the queue empty is not practical because of the variability in packet hand off delays in software. In our experience, some times it may take up to one microsecond longer than usual for the software to hand off a new packet to the NIC. If transport hands off a new packet to the NIC one microsecond after the previous packet has been fully serialized, then the NIC's uplink remains idle during that time and uplink bandwidth is wasted. We found out that if we hand off a new packet to NIC when the NIC's queue length is less than two full-size packets (including the new packet), then this allows us to avoid wasting bandwidth at the uplink, without incurring any significant head of line blocking in the queue.

Chapter 4

Single Receiver Evaluation

In this chapter we evaluate latency performance of the Homa transport design using a network simulator. We focus on the evaluation of the single receiver scenario and the monogamous scheduling as presented in Chapter 3. This allows us to separate out the evaluation of the low latency goal from the high bandwidth utilization goal (which is a topic of later chapters). Hence, we can focus on the effectiveness of the priority assignment and the receiver-driven packet scheduling in achieving preemption at the receiver's TOR queue and ultimately achieving low latency for short messages.

Here we summarize our key findings in this chapter: Homa can achieve 99%-ile tail latency for short messages at about 1.8 times the minimum latency for these messages; this translates to $4\mu s$, 99%-ile latency (one way transmission) for these messages. Homa achieves this latency performance even when the receiver's downlink is 90% loaded. The latency performance of Homa is near optimal; with a limited number of priority levels, Homa achieves very similar tail latency to an ideal SRPT scheduler with an unlimited number of priority levels.

4.1 Homa Simulator Structure

We simulate Homa and evaluate its performance using the OMNeT++ simulator. OMNeT++ is a general purpose, discrete event simulator that has been used in a variety of applications, including wired/wireless network simulations, modeling of queuing networks, modeling of multiprocessors and distributed hardware systems, validating hardware architectures, etc. The simulator's event queue and processing core are written in the C++ programming language. The user, though, can create simulation scenarios, topologies, and configuration inputs either in C++ or a domain specific language called NED. The NED specification is eventually compiled down to C++ source code.

Since the primary use of OMNeT++ simulations is in network simulations, OMNeT++ provides a framework called INET to facilitate network simulations. INET can be considered as the standard network protocol model library of OMNeT++; it contains models for the internet stack (IP, TCP, UDP, etc.), wired and wireless link layer protocols (Ethernet, PPP, IEEE 802.11, etc.), and so on. We built the Homa simulator using OMNeT++ and the INET framework. The Homa simulator consists of several components that we discuss in the following subsections.

4.1.1 Network Topology

Figure 4.1 shows the network topology used for simulations. The network has a full bisection bandwidth topology. It consists of 144 hosts divided among 9 racks with a 2-level switching fabric. For the purpose of experiments in this chapter, one host is the receiver of packets and all other hosts in the network are senders that transmit packets to the receiver. Host links operate at 10 Gbps and TOR-aggregation links operate at 40 Gbps. The simulated switches do not support cut-through routing. Speed-of-light propagation delays are assumed to be 0. The simulations assume that host software has unlimited throughput (it can process any number of messages per second), but with a delay of 1.5 µs from when a packet arrives at a host until it has been processed by software and transmission of a response packet can begin. We chose this delay based on measurements of the Homa implementation. The total round-trip time for a receiver to send a small grant packet and receive the corresponding full-size data packet is thus about 7.8 µs and RTTBytes is about 9.7 Kbytes (this assumes the two hosts are on different TORs, so each packet must traverse four links).

Network traffic is load-balanced at the packet level to minimize queueing in the core of the network in this topology; the switches implement packet spraying [11], so that packets from a given host are distributed randomly across the uplinks to the core switches. Each packet traverses through the network from the sender host to the sender's TOR switch, then transmitted on a random uplink to an aggregation switch. The packet then travels on the only available path from the aggregation switch down to the receiver's TOR and then the receiver host. The only place where packet spraying load balancing happens in this topology is when choosing an uplink from the sender's host to and aggregation switch.

The network provides eight priority queues at each port, which is the same number as the QoS levels that Ethernet infrastructure provides according to the IEEE 802.1Q standard. The ports that haver priority queues include the host NICs' transmit queues, the TOR switches' egress ports, and the aggregation switches's egress ports. The source hosts tag each sent packet with a priority level. The packet is then placed in the priority queue corresponding to the packet's priority tag when it

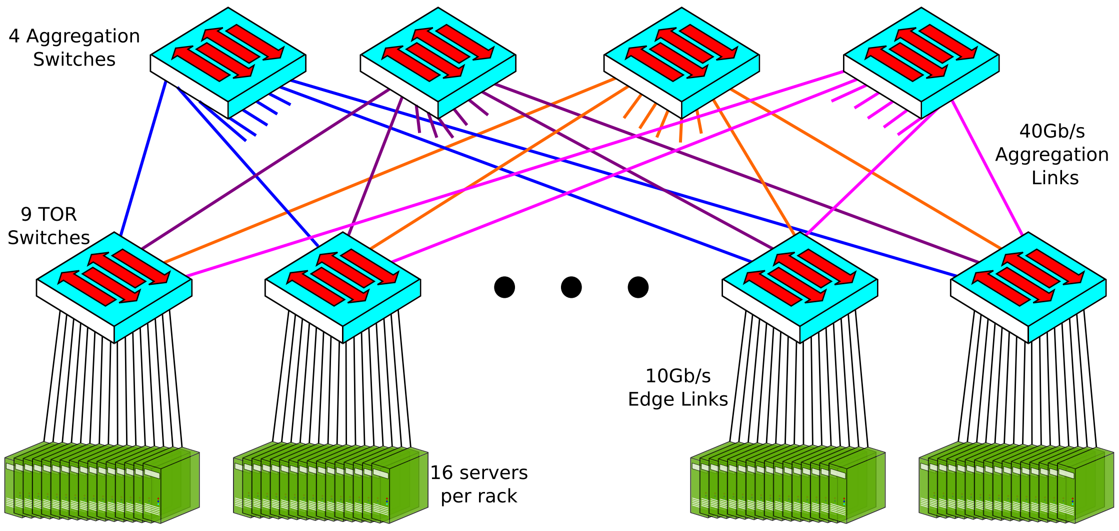


Figure 4.1: A schematic of the network topology used in the simulations. The topology is a two tier, leaf-spine, clos network that provides full bisection bandwidth. The edge links run at 10Gbps and the aggregation links run at 40 Gbps.
The topology also models the realistic delays we have observed in our experimental clusters: 1) Each switch in this topology adds an internal per-packet fixed delay of 250ns before forwarding the packet. 2) Each host models software turn-around times as a delay of 500ns in both of the transmit and receive paths of packets to and from NIC. 3) Each host NIC adds 500ns of fixed delay in the transmit path for each packet before serializing it onto its outbound link.
The RTT in this network is $7.8\mu s$, computed for a full 1538-byte Ethernet data frame that travels through 3 switches in one direction and a 96-byte grant packet that travels in the reverse direction.

arrives at a port. The arbiter logic at each port always prefers forwarding packets from the highest non-empty priority queue; it uses a strict priority policy to schedule packets from the queues for transmission on that port. As in real network switches a packet cannot be preempted once it has begun transmission on a link. As we discussed in Chapter 3, Homa relies on these priority queues to achieve preemption and precisely implement the SRPT policy.

4.1.2 Message Generator

The Message Generator is a component of the simulator that is responsible for creating messages at the sender hosts for a given message rate and also manages the state for generated messages. The message generator accomplishes these responsibilities through two submodules: *Application* submodule and *Workload Synthesizer* submodule. Each host in the network has an instance of these two submodules. The Aplication submodule handles the creation of new messages for transmission at the sender host and maintains the state for each new message until it's completely received at the Application's counterpart submodule on the receiver host. Each Application submodule is associated with a *Workload Synthesizer* submodule that is responsible for generating message sizes and interarrival times between messages. This submodule samples message sizes from a workload distribution measured in real datacenters. It also generates the inter arrival times between consecutive messages from a user specified probability distribution like exponential, Pareto, etc. Given the average message size from the workload distribution, this module scales the interarrival times such that a user-specified average load factor for the sender's NIC link is satisfied.

The Message Generator produces messages in an open loop fashion such that there is no dependency between the generation of two consecutive messages at each sender host; a new message can be generated at a sender host without waiting for the completion of the previous ones. This open loop behavior is different from most application behaviors where there's a kind of application level closed feedback loop (or flow control) that prevents generation of new messages until previous ones are completed and responded by the receiver hosts. In an open loop message generator, there is no limit on the number of active messages from a single sender if the network and the transport can't keep up with transmitting the generated messages (in which case the active message queue at the sender keeps growing without bound). The benefit of an open loop message generator is that it's capable of maintaining an average load on the sender's link to the TOR switch. Therefore we can focus on evaluating the network and the transport mechanism under different network loads, regardless of how the applications behave.

4.1.3 Homa Transport Module

This module is the primary component of the simulator; it implements congestion control logic and packet scheduling, priority assignment, and the scheduling policy. This module is also responsible for the reliable transmission of messages from a sender host to a receiver host. When a sender application wants to transmit a message to a receiver counterpart, it hands over the message in its entirety to the transport module. The transport then packetizes the message and transmits the message over the network to the its receiver counter part. When the transmission is complete and the message is received in its entirety at the receiver transport, it hands the received message over to the Application module on the receiver host.

The transport module consists of two main submodules: The *SendController* and the *ReceiveScheduler*.

***SendController** submodule*

The *SendController* submodule manages the transmission path of the messages from sender applications. Every time a new message is presented from a sender application, this submodule adds the message to the list of the outstanding messages for transmission. This submodule then controls the transmission of the messages in this list. It is responsible for enforcing the transmission policy at the send path, packetization of the messages, and handing the packets over to the NIC for serialization onto the network.

The *Send Controller* manages several important aspects of the Homa transport that we discuss here:

1. *It prepares the unscheduled packets.* When a new message is presented to the transport by the sender application, *SendController* allocates unscheduled packets and encapsulates the first RTTBytes of the message in them. It packetizes the first RTTBytes for sending to the NIC's send queue, but it doesn't send them immediately. Items 5 and 6 below describe the order and the times at which the packets are handed over to the NIC's send queue.

2. *It assigns priority levels for unscheduled packets. SendController* fills out the priority level field in each unscheduled packet header, based on the priority cutoff sizes advertised by the receiver of a message.

3. *It prepares the scheduled packets.* When a grant packet arrives for a message, this submodule creates a new scheduled data packet for that message. It uses the priority level that the receiver specified in the grant to set the priority field in the scheduled packet's header. However, similar to the unscheduled packets, the time and the order in which the scheduled packets are handed over to the NIC is determined by items 5-6 below.

4. *It Implements a line-rate packet pacer for outgoing packets.* This submodule paces packet handover to the NIC transmit queue at line rate; it only sends a new packet to the queue, when the previous packet is fully serialized onto the outbound NIC link (this approach only works in the simulations; in real systems we need to maintain a small queue to avoid bubbles on NIC's outbound link). Keeping the NIC queue short reduces preemption lag and ensures no head of line blocking happens for packets of short messages. This pacer functionality is crucial for a precise implementation of the SRPT policy.

5. *It enforces SRPT among all of the outstanding messages.* Homa uses the SRPT policy to

schedule transmission of messages over the network (refer to Chapter 3 for detailed discussion about SRPT). As described in the previous item on this list, to avoid head of line blocking at NIC's transmit queue, the *SendController*'s pacer only sends a packet to the NIC when the queue has just gone empty. When multiple outstanding messages have packets ready to be transmitted, SendController first checks if the pacer allows a new packet to be sent. When the pacer allows, *SendController* sends the packet from the message with the shortest remaining bytes to the NIC's send queue. This ensures that SRPT policy is enforced by senders.

6. *It manages and prioritizes transmission of control packets.* In addition to the unscheduled and scheduled data packets, each transport sends a few types of control packets. An example of these control packets is the high priority grant packets that the transport sends in the receive path for incoming scheduled data packets from other senders in the network. *SendController* manages transmission of these control packets and prioritizes them over all data packets.

ReceiveScheduler *submodule*

This submodule handles reception of data packets. It is capable of handling out-of-order arrivals of data packets, packing them into full messages, and transferring the messages to the application modules.

The *ReceiveScheduler* submodule implements the majority of the transport mechanisms and logic; it implements Homa's receiver-driven congestion/rate control scheme and the scheduling policy among the incoming messages. The list below covers the specific roles of this submodule in more detail.

1. *It implements the rate control mechanism.* The grant mechanism that is key to Homa's congestion control scheme is implemented by this submodule. This submodule achieves the rate control by sending timely grants for scheduled packets as described in Chapter 3.

   The grant mechanism ensures that the inbound bit rate from scheduled packets of the messages doesn't exceed the receiver's inbound link rate. Hence, it can carefully manage the buffer buildup at the TOR's inbound link to the receivers. Buffers can still build up from the concurrent arrival of unscheduled packets from multiple messages destined to the same receiver. But as we discussed in Chapter 3, this buffer is small and comprised of packets that are preemptable by priorities.

2. *It enforces the SRPT scheduling policy.* The rate control mechanism determines when a new grant can be sent, but SRPT policy determines the order in which inbound messages of the

receiver should be granted. Based on this policy, the *ReceiveScheduler* chooses the message with the shortest remaining bytes to receive and sends a grant packet for that message when the rate control mechanism allows transmission of a new grant.

3. *It records the message size distribution of the current workload.* Using the set of message sizes that has arrived at this receiver in the past, *ReceiveScheduler* generates a cumulative distribution function (CDF) of message sizes of the incoming traffic. This CDF can then be used to devise the priority cutoffs for unscheduled packets as described in Chapter 3.

4. *It allocates priorities for scheduled packets.* Homa receivers utilize priority queues in the network to ensure a precise enforcement of the SRPT policy. For scheduled packets, the *ReceiveScheduler* adaptively changes the packets' priority levels based on the current set of inbound messages to implement SRPT; the details of this scheduled priority assignment scheme were explained in Chapter 3. In summary, whenever a new message is added to the set that becomes the new highest priority message, *ReceiveScheduler* uses the next higher priority level (if one is available) for the scheduled packets of that messages and grants that message on the higher priority level.

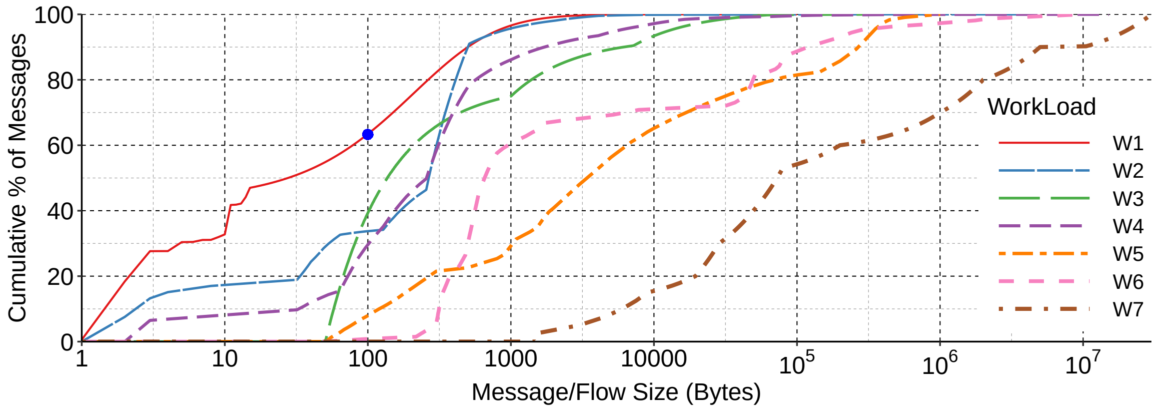
5. *It calculates the priority cutoffs for unscheduled packets. ReceiveScheduler* uses the measured CDF of the traffic workload and computes priority cutoffs to enforce SRPT policy. In a nutshell, *ReceiveScheduler* allocates unscheduled priorities among the messages of different sizes in a way that shorter messages receive higher unscheduled priority levels. *ReceiveScheduler* then transmits the computed cutoffs back to the senders by piggy backing them on grant packets. The senders use these cutoffs to set priority of unscheduled packets. The details of this scheme were explained in Chapter 3.

4.2 Workload Distributions

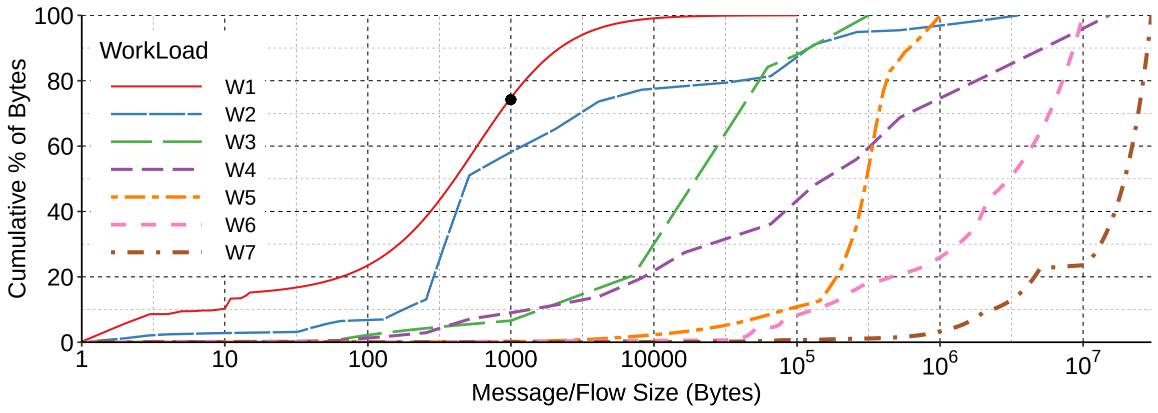
For evaluating Homa, we used a set of seven workloads, most of which were measured in real datacenters. These workloads were briefly discussed in Chapter 2 but we represent them here again and explain them in more detail. Table 4.2(a) contains descriptions for these workloads. All the workloads in this set, except W3, are measured from production traffic in Google, Facebook, and Microsoft datacenters.

W1	Accesses to a collection of memcached servers at Facebook, as approximated by the statistical model of the ETC workload in Section 5 and Table 5 of [6].
W2	Measured from a search application workload at Google [33].
W3	A synthesized workload.
W4	Aggregated workload from all applications running in a Google datacenter [33].
W5	Measured from intra cluster traffic of Web Server applications at Facebook [31].
W6	Aggregated workload from all Hadoop traffic at Facebook[31].
W7	Web search workload used for DCTCP [3].

(a)



(b)



(c)

Figure 4.2:
a) The workloads used to design and evaluate Homa.
b) The upper graph depicts the cumulative distribution of message sizes weighted by number of messages. The blue point on the upper graph means that messages that are smaller than 100Bytes account for 63% of all messages in W1.
c) The bottom graph shows the cumulative distribution of messages weighted by transmitted bytes. The black point on the bottom graph shows that in workload W1, messages smaller than 1000Bytes, account for 75% of the traffic bytes transmitted over the network.
Workloads W1, W2, W4 were measured from application-level logs of message sizes; message sizes for W5, W6, and W7 were estimated from packet traces. For the sake of completeness, W3 is synthesized to cover the gap between W2 and W4 in the bottom graph. The workloads are ordered by average message size: W1 is the smallest (i.e. the most heavy-head), and W7 is the most heavy-tailed workload.

We simulated the performance of Homa under each of these seven workloads. In a single simulation run, we choose one of these workloads. Each sender host then generates message sizes that are sampled from the probability distribution function of message sizes for that workload. Figure 4.2(b) shows the cumulative distribution function of message sizes for each of these seven workloads.

We picked these seven distributions to be representative of the whole traffic spectrum of possible workloads in datacenter networks. Figure 4.2(c) shows the cumulative percent of bytes transmitted for different message size of each workload. The plots in this figure illustrate the distribution of network traffic bytes over different ranges of message sizes for each workload. The plots of the seven workloads in the figure span a large area from far left to the far right of the plot. For example, on one extreme of the spectrum is the heavy-tailed W7 workload, the rightmost plot on the figure. For W7 the vast majority of network traffic bytes are transmitted in very large messages (98% of the traffic is in messages larger than one megabyte). On the other hand, the W1 plot is located on the other extreme of the spectrum to the left of the figure and most of the network traffic bytes of this workload are transmitted in short messages (less than 100bytes); about 80% of network traffic bytes in workload W1 are transmitted in messages that are shorter than 1000 bytes in size. The other six workloads cover the space between the two extremes of W1 and W7 on the figure to span the full spectrum of possible realistic workloads in modern datacenters. In later sections we show how well Homa achieves its design goals for all these workloads.

For the purpose of this thesis, we refer to workloads similar to W1 as heavy-head workloads. In the literature, workloads like W7 on the right side of the spectrum in Figure 4.2(c) are commonly referred to as heavy-tailed workloads. That's because most of the network traffic bytes are transmitted in large messages at the tail of the spectrum. Both W6 and W7 are considered to be heavy-tailed. In contrast, we refer to workloads W1, W2, and W3 as heavy-head workloads to make the distinctions that most of traffic bytes are transmitted in short messages (i.e. less than one full packet). We'll show how different mechanisms in the design of Homa benefit both heavy-head and heavy-tailed workloads and any other workload in between.

4.3 Homa Performance Evaluation

In this section our goal is to evaluate the unscheduled priority mechanism of Homa and see how it performs under heavy unscheduled load. To examine this mechanism, we focus on the single receiver, "monogamous design" of Homa as presented in Chapter 3. Given the network topology of Figure 4.1, where hosts are indexed from 0 to 143, we assume that one of the hosts (e.g. host 0) is the single receiver in the network and the other 143 hosts send messages to that receiver. Since low latency at the tail is the main goal for Homa, this chapter uses the tail latency as the main metric to evaluate Homa's unscheduled priority assignment scheme. We'd like to show that the combination of Homa's unscheduled priority assignment and the receiver-driven congestion control scheme achieves very low tail latency.

In each simulation experiment, our goal is to sustain an average network load on the receiver's downlink. In any individual experiment, we choose one of the workloads and each sender generates messages that are randomly sampled from the CDF of message sizes for that workload. Each sender then generates interarrival times between messages from a Poisson distribution, with average interarrival times set to maintain a specific network load on the link to the receiver. For example, if the goal of the experiment is to maintain the receiver's downlink at 80% network load, then each of the 143 senders generate interarrival times from a Poisson distribution such that each sender's uplink is $\frac{80\%}{143}$ loaded.

In order to evaluate Homa's unscheduled priority assignment scheme, in this chapter we only use the subset of workloads that can create significant unscheduled traffic. We use Figure 4.3 to choose which workloads belong to this subset. On this figure, we can see that only W1 to W4 can generate significant unscheduled traffic; from W1 where almost 98% of the traffic comes from unscheduled packets, to W4 where about 30% of the total traffic on the network is from unscheduled bytes. In the other 3 workloads, the unscheduled traffic is insignificant. Hence, in this chapter we focus on evaluating unscheduled priority assignment scheme for W1 to W4. The remaining 3 workloads will be considered in the future chapters where we evaluate other aspects of Homa's design that are related to scheduled packets.

4.3.1 Slowdown: Latency Metric of Our Choice

The interesting latency question to ask is, how much slower is a message transmitted under a moderate or high network load, compared to when the message is transmitted in a completely unloaded network? The answer to this question is the definition for the **slowdown** metric. Slowdown of a message with a certain size is defined as the measured latency of the message in a simulation, divided by the minimum latency for that message in an unloaded network. For any message, the optimal slowdown is one and higher slowdown values are worse because higher slowdown means the message has experienced higher latency compared to its minimum latency. For all simulations results in this dissertation, latency is measured one-way for a message from when the message is handed to

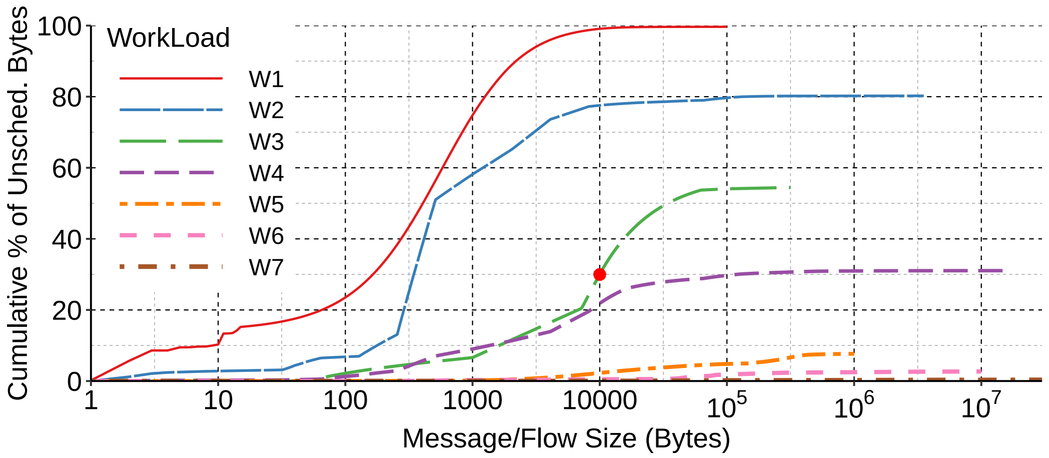


Figure 4.3: Cumulative distribution of unscheduled bytes as a function of message sizes for each workload. Each distribution is normalized as a fraction of total bytes transmitted over the network. 100% on the y-axis corresponds to all network traffic, both scheduled and unscheduled; in W2 about 80% of all traffic is from unscheduled bytes. For example, the red point on the plot shows that unscheduled bytes for messages shorter than 10Kbytes, account for 30% of the total bytes transmitted over the network in workload W3.

the sender's transport until it is fully received at the receiver's transport. Throughout this dissertation, we use median, mean, and tail slowdown instead of raw latency numbers in our performance measurements.

In order to study the slowdown of messages under Homa, we introduce a new kind of plot that we call "slowdown spectrum". This type of plot depicts a metric of slowdown such as tail slowdown for different message sizes of a specific workload, and at the same time, allows us to focus our attention on the performance of the messages that are the most frequent in the workload distribution. Figure 4.4 shows an example of a slowdown spectrum plot. This figure plots 99%-ile tail slowdown on the y axis vs message sizes for workload W4 on the x axis. However, the x axis is linear in quantiles of the messages in W4; i.e. the axis is scaled by the CDF of message sizes for W4, with each x tick corresponding to to 10% of all messages. For example, the blue dot at the 4th x-tick on the plot shows that messages smaller than 158 bytes account for 40% of all messages in W4 and 99%-ile tail latency for a 158-byte message is at most 1.7 times the minimum latency for that message.

The scaled x-axis on a spectrum plot allows us to focus on the performance of Homa for the most important messages of each workload; i.e. messages that are the most frequent. Each workload can have any message size ranging from a few bytes to tens of megabytes. However, based on

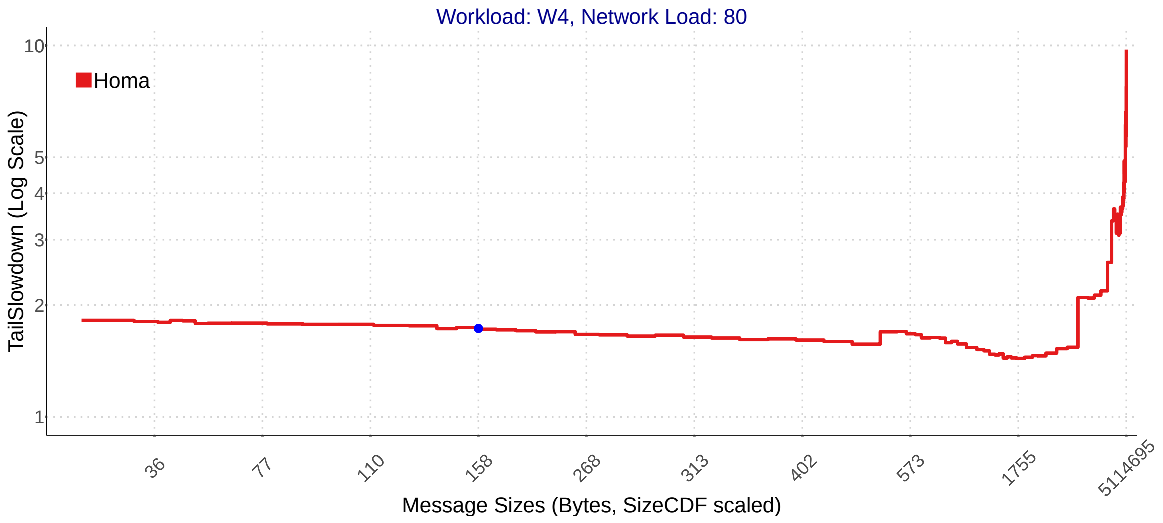


Figure 4.4: A sample slowdown spectrum plot, the primary type of latency plots that we use throughout this dissertation. This figure shows the latency results of a Homa simulation with workload W4, at 80% load on the receiver's downlink. The x axis shows message sizes in workload W4 and the y axis shows 99-%ile tail slowdown (i.e. normalized latency) for different message sizes. The x axis is is scaled by the CDF of message sizes for W4 (i.e. it is linear in the number of messages transmitted), with each x tick corresponding to 10% of all messages. The benefit of this scaling of the x axis is that, for example we can see that for more than 90% of all messages in W4 (i.e. 1755-byte and less on the x axis), the 99% tail latency is less than two times the minimum latency for each message.

the CDF of message sizes for a workload, messages of a certain size might be more common than other messages in that workload and therefore they deserve more attention in protocol design and performance evaluation. The benefit of scaling the x axis with message quantiles is that it presents information about how frequent various message sizes are in the workload and highlights the slowdown results for the most frequent messages. For example, in Figure 4.4 we can see that messages smaller than 1755-byte, are highly frequent as they account for 90% of all of messages in W4. Moreover, the 99%-ile slowdown for all these messages is less than 2 which, as we see later in this chapter, is very close to optimal tail latency.

Note that in a slowdown spectrum plot, the x axis can either be linear in the number of messages or be linear in the number of transmitted bytes in the network. In Figure 4.4, the x axis is scaled by the CDF of message sizes (i.e. CDFs of Figure 4.2(b)) and the axis ticks represent quantiles of message counts. So, we refer to this spectrum plot as ”count spectrum”. Conversely, the x axis may be scaled by the CDF of bytes transmitted (i.e. CDFs of Figure 4.2(c)). In this case the x axis ticks represent quantiles of transmitted bytes for the workload. Based on the CDF of bytes for a workload, messages of a certain size might be responsible for carrying more traffic bytes compared to the other messages and therefore they might deserve more attention. Scaling the x-axis by the CDF of bytes transmitted, allows us to turn the focus of our evaluation to the message that carry majority of traffic bytes in the network. We refer to this type of spectrum plot as ”byte spectrum”. Throughout this dissertation we use both of these spectrum plots to explain the latency performance of Homa.

4.3.2 Homa’s Slowdown Performance

In the first set of performance evaluations, we study the tail slowdown when the receiver’s downlink is highly loaded in simulations. Figure 4.5 shows 99%-ile tail slowdown vs message sizes for workloads W1 to W4 when the receiver’s downlink is 80% utilized. The plots on the left column of the figure represent the count spectrums for the workloads and the ones on the right show the byte spectrums. The red curves on the plots show Homa results. In all plots, regardless of the workload type, tail slowdown increases as the message size increases. This behavior is a result of the SRPT policy implementation for Homa; when congestion happens at the TOR, the high priority packets that belong to shorter messages are favored over the low priority packets of larger messages and short messages experience lower latency.

On the plots of Figure 4.5, we compare Homa’s simulation performance against a ”PseudoIdeal” transport that can achieve near optimal latency for short messages in our single receiver network. However this transport is impractical because it requires an infinite number of priority levels from the network fabric, with queues having infinite capacity. In this PseudoIdeal transport, each sender transmits packets at line rate. Each packet is tagged with the size of the message it belongs to. At each queue in the network, the packets are ordered based on the size of the message it belongs to, with packets of shorter messages closer to the head of the queue. Queues are assumed to have infinite capacity and ordering ties in the queues are broken with the arrival times of the messages at the senders. Since the network traffic matrix is n to 1 (i.e. 143 senders to 1 receiver), the primary congestion point in the network is TOR queue at the receiver’s link, and in this queue, packets of shorter messages are always prioritized. Hence, in an n to 1 network, absolute low latency is achieved for short messages with this PseudoIdeal transport.

From Figure 4.5, we see that Homa can achieve tail latency close to the near optimal performance of the PseudoIdeal transport, but Homa does that with a limited number of priority levels. While the PseudoIdeal transport relies on an infinite number of priority queues in the network, Homa (depending on the workload type) in these evaluations only requires up to 6 priority queues from the network switches. The count spectrum plots show that the majority of the transmitted messages are

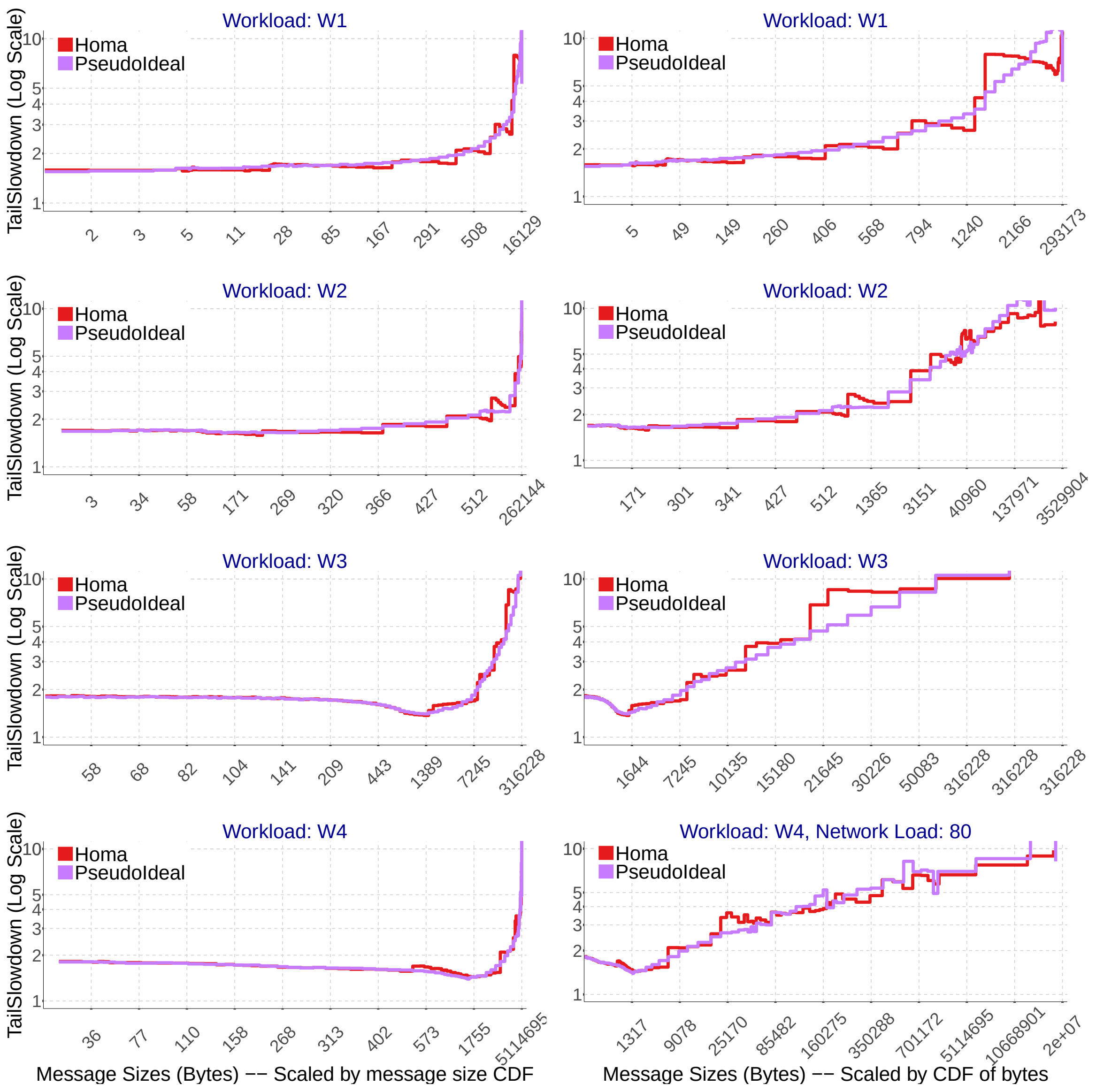


Figure 4.5: Performance of Homa's unscheduled priority assignment scheme for workloads W1-W4 when receiver's link is 80% loaded. The two plots on each row show 99%-ile tail slowdown vs. message sizes for one of the workloads. The count spectrum plots are in the left column and the byte spectrum plots are in the right column. Regardless of the workload, Homa achieves tail slowdown of less than two for more than 90% of all messages. Each plot, in addition to Homa's results, also shows the results for the PseudoIdeal scheme as a reference for the best possible performance. In general, Homa achieves near optimal performance even at this high 80% load.

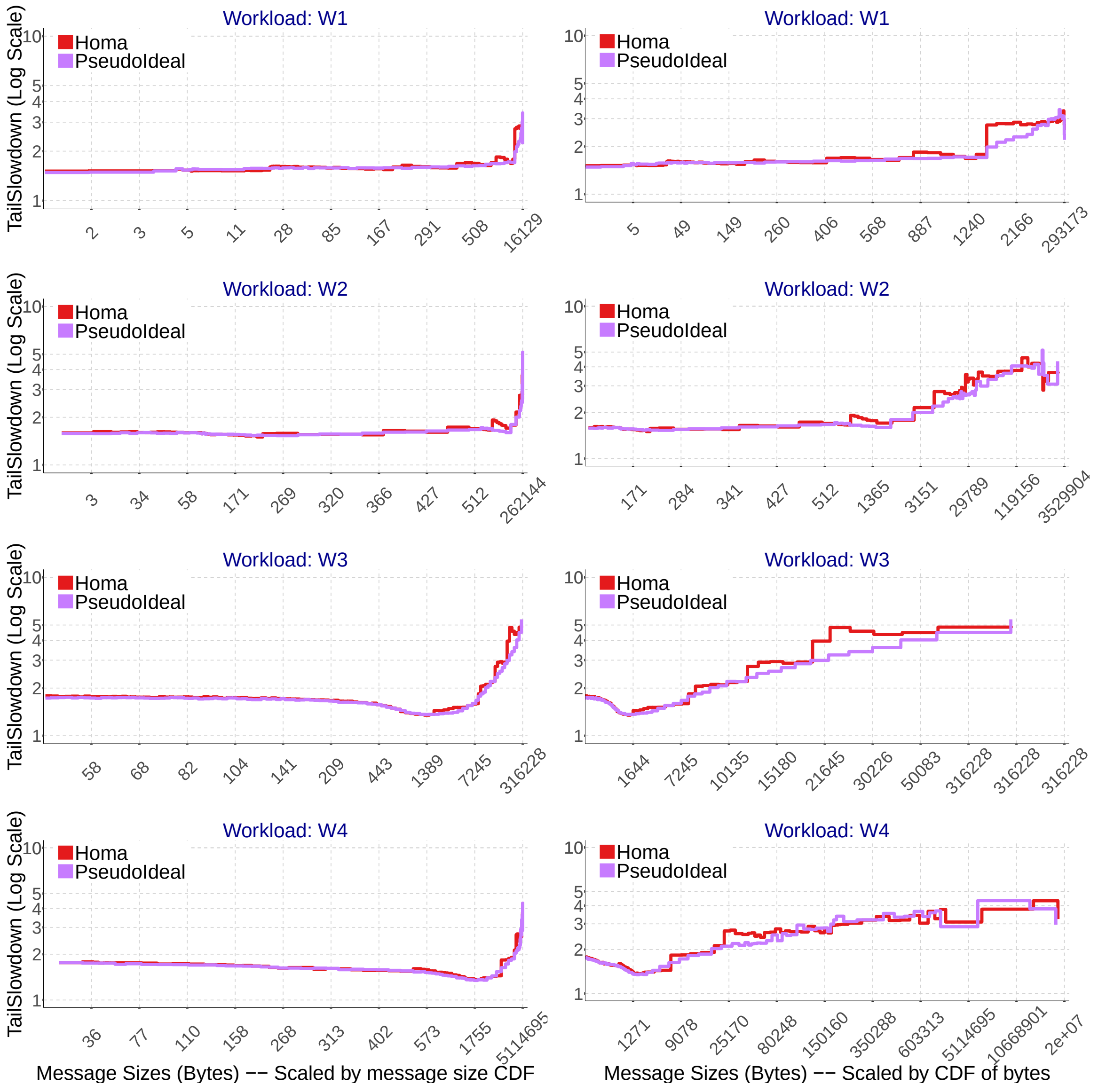


Figure 4.6: Similar to the plots of Figure 4.5 but at medium 50% load on the receiver's downlink. Both Homa and PseudoIdeal scheme achieve lower (i.e. better) tail slowdown at lower load factors.

experiencing tail latencies that are close to the PseudoIdeal's performance within a few percent. The byte spectrum plots similarly reveal that messages that carry the majority of traffic on the network, also experience near optimal tail latencies within a few percent of the PseudoIdeal performance.

When we approximate SRPT with a limited number of priority levels, the largest slowdown deviation from PseudoIdeal happens for the group of messages with the lowest priority. Furthermore, among all messages in that group, the ones with the shortest size experience the highest deviation. This slowdown deviation from PseudoIdeal manifests itself on the byte spectrum plots of Figure 4.5, around message size 1240 bytes for W1 and 21645 bytes for W3; the jumps on the Homa curves around these message sizes happens because these message sizes correspond to unscheduled priority cutoffs associated with the lowest unscheduled priority levels. Specifically, all messages larger than 1240 bytes in W1 and all messages larger than 21645 bytes in W3 are sending unscheduled packets at the lowest unscheduled priority level. Among all of these low unscheduled priority messages, the shortest of them experience the highest slowdown overhead. It is important to note that from count spectrum plots, it's clear that these high slowdown messages account for a tiny fraction (less than 2%) of all messages in W1 and W2.

Reducing the load on the receiver downlink from 80% to a lower load reduces the tail slowdown. Figure 4.6 is similar to Figure 4.5, but uses a moderate 50% load rather than 80%. The figure shows that regardless of the workload type, the tail latencies have reduced over the whole spectrum of message sizes in comparison to the plots at 80% network load. At 50% load there is shorter queues in the network compared to the 80% load and packets will experience less queuing delay, hence lower tail latency. Moreover, at lower loads, Homa achieves performance that is closer to optimal performance; the difference in performance between Homa and PseudoIdeal reduces at lower loads.

4.3.3 Unscheduled Priority Allocation Scheme

One of the important questions we need to address is how did we arrive at the design for Homa's unscheduled priority scheme as described in Chapter 3? When we were developing Homa's priority allocation scheme we considered several designs. We empirically evaluated these designs and arrived at the solution that we discussed in Chapter 3 because it achieved the lowest latency for short messages over various workloads. In this section, we discuss three of the schemes we considered and empirically show that Homa's scheme achieves better latency compared to the other two schemes. We use the empirical approach for comparison because we are not aware of any analytical or theoretical tools to analyze the optimality of Homa's design. Below we briefly explain Homa's scheme and the other two approaches, then we present and analyze the performance of each approach.

Unscheduled priority assignment scheme in Homa approximates SRPT with a limited number of unscheduled priority levels. To that end, this scheme tries to balance the unscheduled bytes between priority levels and assigns priority levels to the unscheduled packets of a message such that the later packets of the message receive higher priority levels than the earlier ones (there are fewer bytes remaining to be transmitted). Furthermore, this scheme applies to both the first RTTBytes of the message and the last RTTBytes of the message. The scheme is explained in detail in Chapter 3 and depicted in Figure 3.7(b). For the purpose of this section, we name Homa's scheme "graduated" priority allocation mode since the unscheduled packets of a single message *gradually* receive higher priority levels as the remaining bytes of the message decreases.

In the first unscheduled priority allocation scheme we compare against, all unscheduled packets of an individual message uniformly receive the same priority level. So we call this scheme "uniform" allocation mode. This is in contrast to the graduated scheme where the later unscheduled packets of a message receive higher priority level. This uniform scheme uses the CDF of transmitted bytes to assign unscheduled packet priority levels such that an equal number of bytes is transmitted over each priority level and the packets of shorter messages receive higher unscheduled priority levels.

The second scheme we compare against allocates an equal number of messages per unscheduled priority level; this is in contrast to the previous schemes that allocated an equal number of bytes per priority level. Suppose that there are k priority levels available for the unscheduled packets. This scheme uses the CDF of message sizes to divide the whole message size range into k size ranges, such that each range holds equal fraction ($\frac{1}{k}$) of all messages. This scheme then uses the highest unscheduled priority level for the range with the shortest message sizes and respectively lower priority levels for the larger size ranges. We call this scheme "CDF Uniform" because all unscheduled packets of a given message uniformly receive the same priority level and priorities are allocated based on the CDF of message sizes.

Figure 4.7 shows the count spectrum plots for the three allocation modes we discussed, along with the PseudoIdeal plot as the ground reference. All plots were generated at 80% network load on the receiver's downlink. The left column in the figure shows W1 to W4 results when there are two priorities available for unscheduled packets and the right column plots in the figure shows the comparison when four unscheduled priorities are available. We restricted the number of priority levels to two and four in this experiment since the difference in the performance is more observable when have fewer priorities.

In Figure 4.7, we see that the CDF uniform scheme has the largest deviation from the PseudoIdeal for the largest fraction of messages and it's the least suitable among the three schemes. Regardless of the priority assignment scheme, when we have two unscheduled priorities, there's a sudden increase in the tail slowdown plots for all workloads and schemes. This sudden jump in tail slowdown happens because messages before the jump use the higher priority level and messages after the jump use the lower priority level and incur a high tail latency hit. In particular for the CDF uniform scheme, this sudden jump happens at 50%-ile of message sizes of workload. This causes

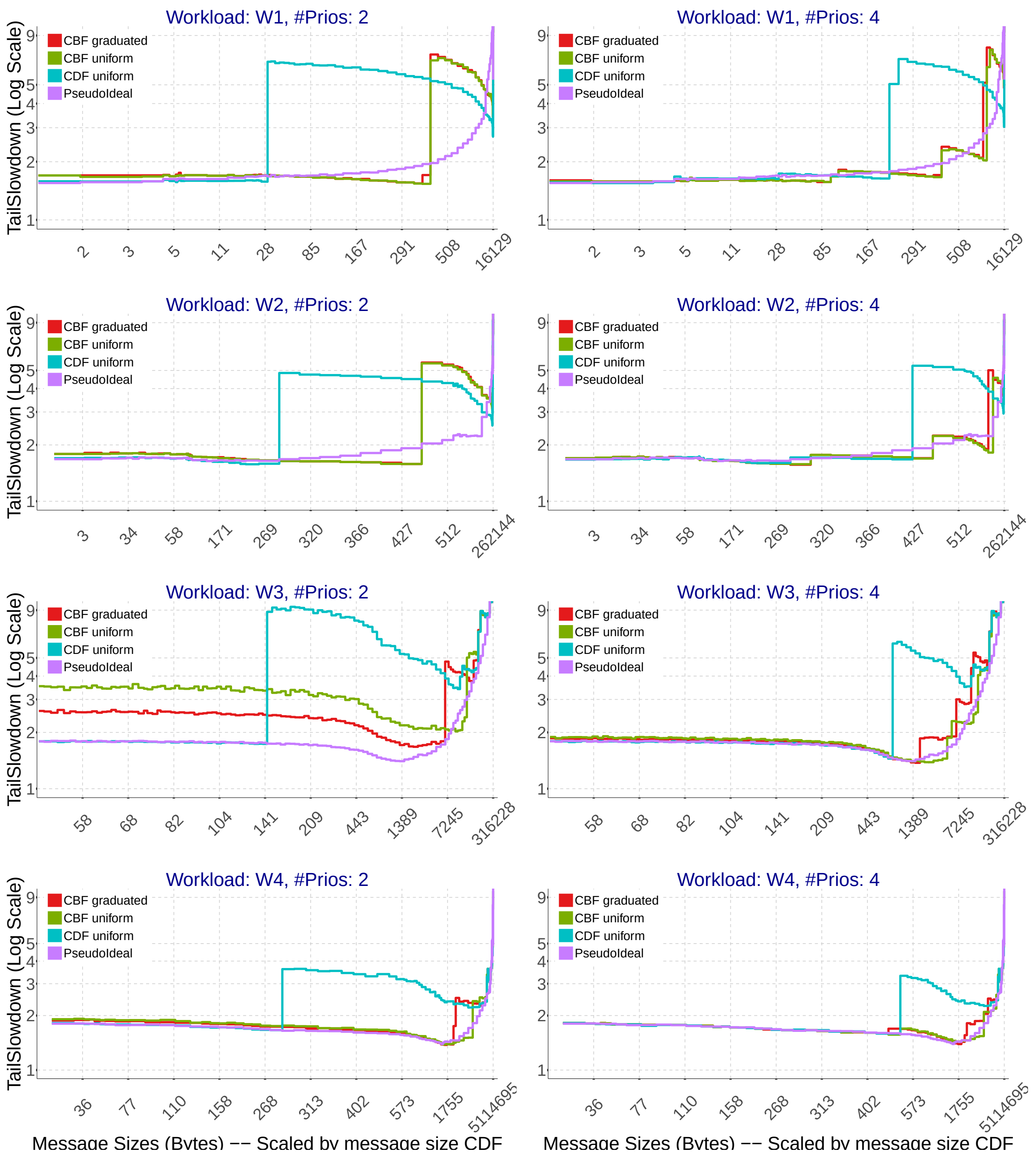


Figure 4.7: Comparison of three unscheduled priority assignment schemes for Homa at 80% network load on the receiver downlink for workloads W1-W4. The plots on the left show the results when there are two unscheduled priority levels, while the plots on the right are for four unscheduled priority levels. The plots show that CDF Uniform is not suitable; compared to the other two allocation schemes, it yields a larger 99% tail slowdown over a large fraction of message sizes for all workloads. This large tail slowdown happens for either two or four priority levels.
CBF Uniform achieves performance close to the graduated scheme in most cases. With two priority levels, the uniform scheme leads to slightly higher slowdown for some messages, but slightly lower slowdown for others.

the CDF uniform scheme to have the largest deviation from the near optimal tail slowdown of PseudoIdeal. Increasing unscheduled priorities to four improves the tail latency of messages by pushing the sudden jump more to the right of the plot at 75%-ile of message sizes in each workload. This sudden jump happens because the largest 25% of message sizes are assigned the lowest priority level among the four available priority therefore they sustain the largest hit on the tail latency.

The spectrum plots of Figure 4.7 are not conclusive which of the two other schemes (graduated or uniform) is better. Over the majority of message sizes for various workloads, the two schemes lead to very similar tail slowdown, with occasional differences where sometimes for a specific message size, one scheme or the other has better tail slowdown. For example, with workload W3 and two unscheduled priorities, the graduated scheme has lower tail slowdown for message sizes shorter than 7245 bytes, while for messages slightly larger than 7245 bytes, the CBF uniform scheme has lower tail slowdown. Hence we need a way to aggregate the results in more compact form to more easily evaluate them against each other.

In order to better understand the similarities between the graduated and CBF uniform schemes, we use a simplified aggregated metric over broader message size ranges. For each workload, we divide the entire spectrum of message sizes into five ranges and compute the aggregated tail slowdown for each range by computing the area under each curve of the Figure 4.7 for each of the five ranges. Figure 4.8 shows the aggregated slowdown metric over the five message size ranges for all workloads and three priority allocation schemes, computed from figure 4.7. Below, we introduce the five message size ranges on Figure 4.8 and explain what motivated us to treat each range as a separate aggregation group. We define the aggregated slowdown for each message size range as the area under the slowdown spectrum for that message size range; i.e. sum of all tail slowdown in the range but weighted by the probability of the message sizes in that range.

1. Super short messages that are smaller than 300B in size. We separate these messages into their own category because on a realistic 10Gbps network topology, the serialization delay for these messages is negligible comparing to the fixed switching and forwarding delays in the network. So the total delay for transmitting these message over the network is dominated by fixed transmission delays rather than serialization delay. This category of short message sizes comprise 60% to 82% of all messages in the W1 to W4 workloads.

2. Short messages that are larger than 300B and no longer than one full packet. In Homa's context, these messages are also considered short messages but the serialization time for these messages is comparable to the network fixed delays.

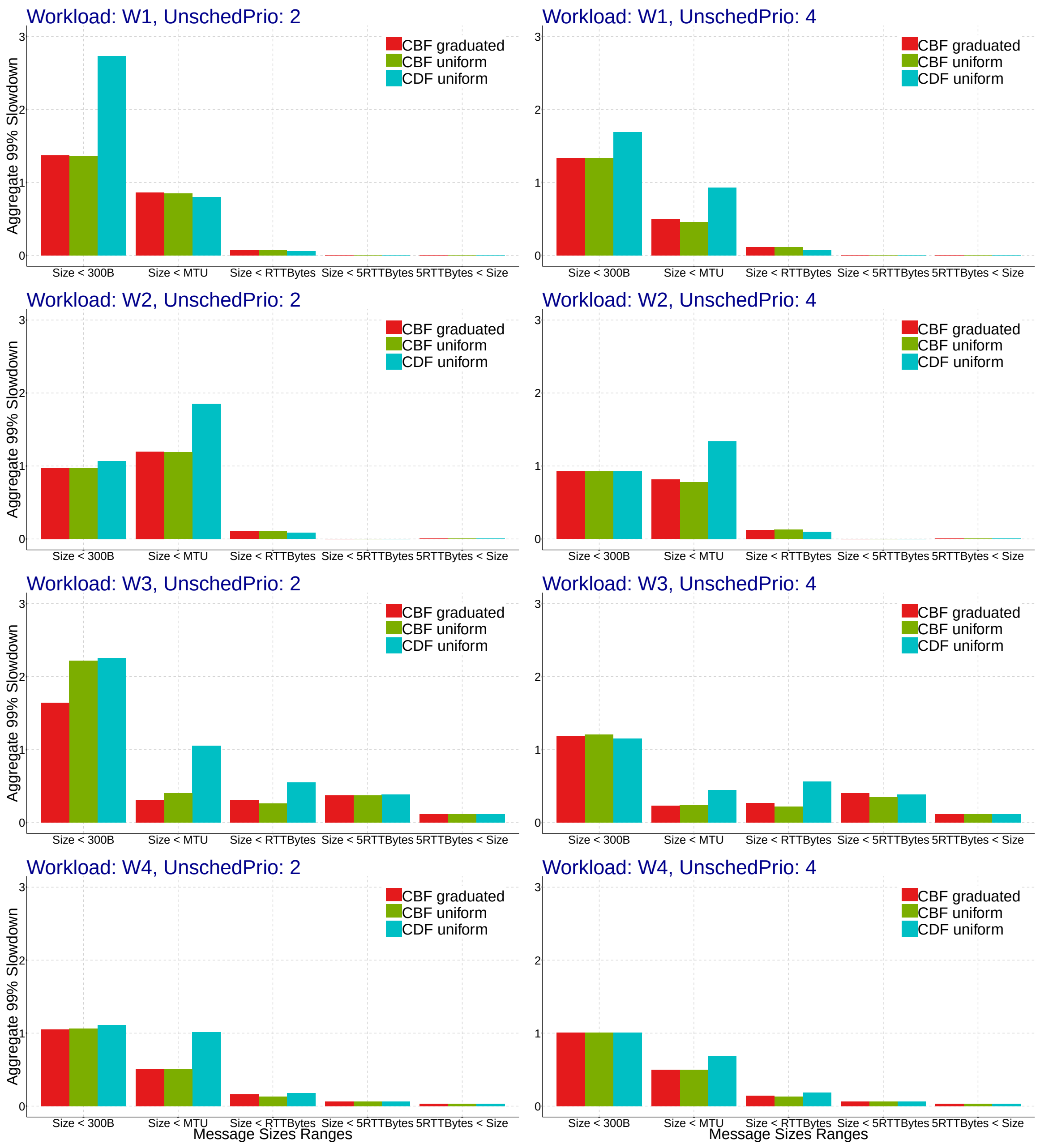


Figure 4.8: Aggregate tail slowdown for ranges of message sizes for three unscheduled priority assignment schemes. Each graph on this figure corresponds to one of the graphs of Figure 4.7. Aggregated slowdown for each message size range is the sum of tail slowdowns for that range in Figure 4.7, weighted by the probability of messages for that range; the aggregated values can be smaller than one if the probability of the range is too small. The probability weighting reflects the importance of the messages in the range; the more frequent messages in the workload are more important to be considered in the evaluation.

3. Medium size messages, longer than one MTU and shorter than one RTTBytes. These messages are distinct in their own group because no scheduling is required for them (they completely fit in unscheduled packets).

4. Large messages longer than one RTTBytes and shorter than 5×RTTBytes. These are considered large scheduled messages that have significant unscheduled portions as well.

5. Super large messages that are larger than 5×RTTBytes in size and the unscheduled portion of these messages is practically negligible compared to the total length of the message.

From Figure 4.8 we can infer that the difference between the graduated and uniform schemes is small and one may argue that the uniform scheme can be used for Homa without significant performance loss. The graduated scheme of Homa has better slowdown for short messages, especially for W3 with two priorities for super short messages. But for most of the other message size ranges and workloads, there's virtually no difference between the results from the two schemes. The difference is even less significant when we have four unscheduled priority levels. Since Homa expects at least four unscheduled priorities for W1 to W4 for the best performance, we argue that in Homa's implementation we may choose to use the CBF uniform scheme instead of the near optimal graduated scheme. This choice of the uniform scheme immediately simplifies the implementation of the unscheduled priority allocation scheme. This is because the CBF uniform scheme only relies on computing the CDF of the received bytes. In contrast, the graduated scheme implementation requires computation of a transformation of the CDF and this computation may cost a lot of cpu time. As a matter of fact and for the sake of simplicity, we use the CBF uniform scheme in our own system implementation of Homa.

4.3.4 Varying Number of Unscheduled Priorities

A question that we should answer is how does the number of available unscheduled priorities affect Homa's performance under different workloads and load factors? Or in other words, how many unscheduled priority levels does Homa require for each workload at various load factors, in order to achieve close to optimal performance? To answer this question we ran simulations at different network loads/workloads and investigated how the slowdown changes as the number of priorities varies. Figure 4.9 shows the results of the simulations. Each plot on this figure shows the count spectrum for either workloads W1 or W3 at one of three receiver's downlink loads: low (30%), medium (55%), and high (80%). We show the two workloads for which the spectrum plots seem to be the most sensitive to the number of unscheduled priorities. On each plot, we have varied the number of available unscheduled priorities from 1, to 2, 4, and 6 and we compare against the PseudoIdeal reference curve. The unscheduled priorities in these plots are in addition to the scheduled priorities. For W1, we use one extra priority level for scheduled packets, and for W3 we use three extra scheduled priorities where the unscheduled traffic is transmitted over the higher priority levels, and the scheduled traffic is transmitted on the lower one.

If we compare one priority curve to the PseudoIdeal curve on each of the plots of figure 4.9, it's immediately obvious that one unscheduled priority is not enough for near optimal tail slowdown. Regardless of the loadfactor on the network, and the workload type, the one priority curves have 1.3–5 times worse slowdown compared to the PseudoIdeal curve. Therefore, it is important to allocate more than one unscheduled priority level for workloads with high volume of unscheduled traffic (i.e. W1-W4), even at low network loads.

Two unscheduled priorities seems to be enough for low network loads, but not for moderate to high network loads. With two unscheduled priorities the tail slowdown is 0.8–3 times lower than one unscheduled priority curves, for most of the message sizes. More specifically at 30% network load, two unscheduled priority levels brings the tail slowdown close to the PseudoIdeal performance on W1 plot and within 20% of the PseudoIdeal on W3 plot. Therefore one may argue that even two unscheduled priorities would be enough to achieve close to optimal tail slowdown for almost all practical workloads.

Figure 4.9 shows that at medium to high network load, four or more unscheduled priorities is enough to achieve close to optimal tail slowdown for all workloads. With four priorities and at 50% network load, Homa's performance is almost indistinguishable from PseudoIdeal tail slowdown. Even at 80%, with four unscheduled priority levels, the tail slowdown is very close to optimal PseudoIdeal performance except for a small fraction of messages around 10Kbytes in W1. With 6 unscheduled priority levels, Homa's performance is almost indistinguishable from PseudoIdeal even at 80% network load. That said, one may argue that with four unscheduled priority levels, Homa's performance is close enough to the PseudoIdeal's performance even at 80% network load.

4.4 Chapter Summary

In this chapter we introduced the Homa simulator and its different modules and we discussed the structure of our simulation experiments in details. We also introduced the network topology and workloads that we use in our simulations throughout this dissertation.

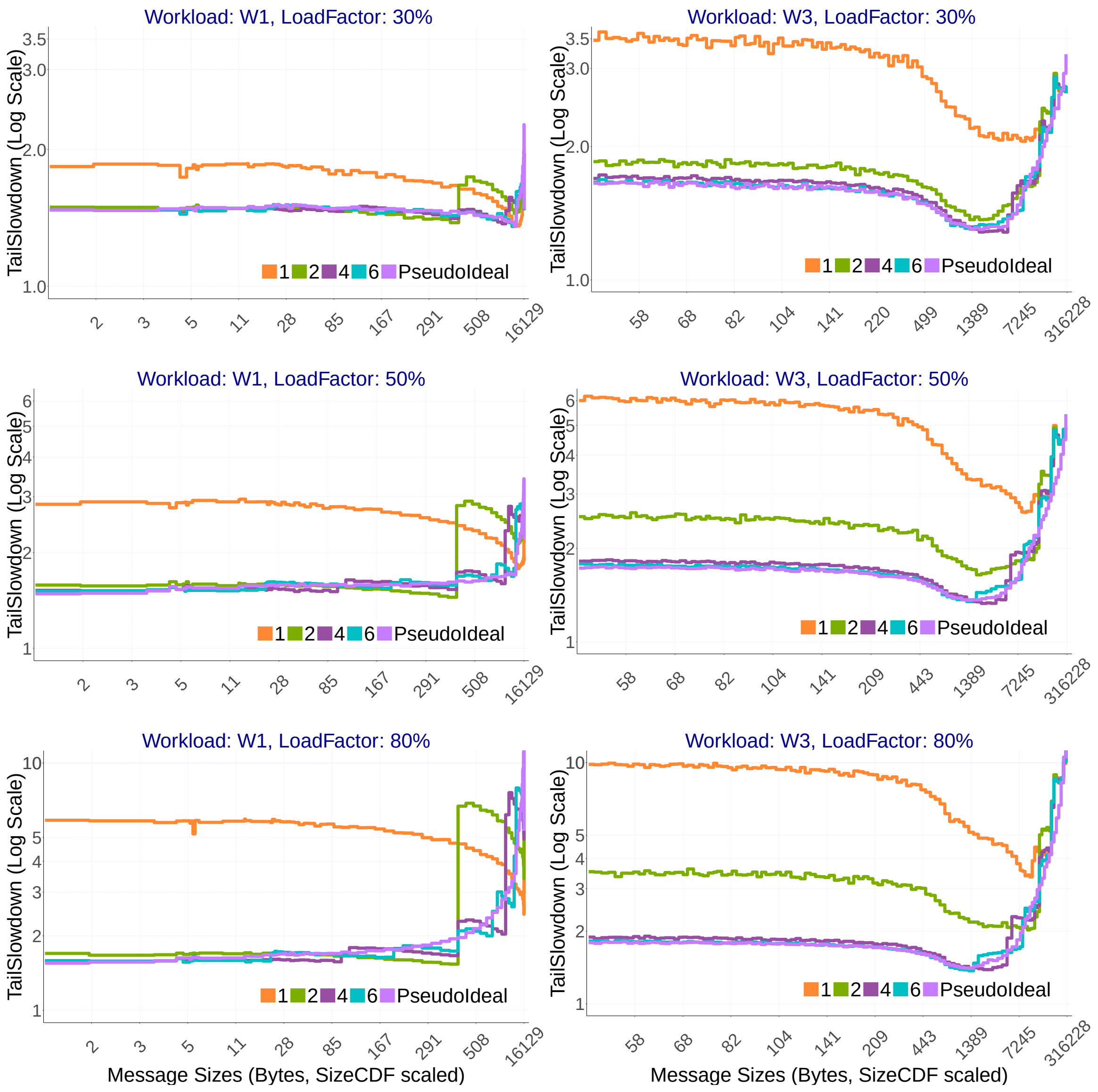


Figure 4.9: Effect of number of unscheduled priority levels on tail slowdown for workloads W1, and W3 at 30%, 55% and 80% load factors. The figure shows that low 30% load factor, Homa needs at least two unscheduled priority levels (in addition to scheduled priority levels) to achieve close PseudoIdeal tail slowdown. At moderate to high load factor (i.e. 50% to 80%), Homa requires four unscheduled priority levels to achieve close to PseudoIdeal tail slowdown.

We introduced the slowdown spectrum plots and discussed how nicely we can depict the latency for different messages and at the same time target our focus on the most important and most frequent messages in the workload.

We evaluated Homa's unscheduled priority assignment scheme and showed that Homa can closely approximate near optimal SRPT performance of the PseudoIdeal scheduler. Homa is able to achieve single digit tail latency for short messages; in our simulations 99%-ile tail latency for short messages is about 4 μs, which is at most 1.8 times the minimum latency for these messages.

We empirically compared Homa's unscheduled priority allocation scheme against two other designs (CBF uniform and CDF uniform). We showed that even though Homa's scheme has better performance, the difference between Homa's scheme and CBF uniform is small. Therefore, one may choose to use the CBF uniform for unscheduled priority assignment since it is easier to compute.

We also showed that one unscheduled priority is not enough to achieve near optimal latency with Homa, even at low network load. However, at low network load, two priority levels should be enough to achieve tail latency within 20% of the optimum for short messages in heavy-head workloads. At higher network load, four priority levels is enough for unscheduled packets to achieve near optimal latency.

Chapter 5

Multi Receiver Design

5.1 Single Receiver Design: Good and Bad

In chapters 3 and 4, we discussed the monogamous design and evaluation of Homa. In those chapters, one of the primary assumptions was that there are many sender hosts and only one receiver host in the network. So, any message from the senders was destined to the single receiver.

We referred to the design in previous chapters as the monogamous design of Homa because each receiver allows only one active message at a time, by scheduling packets from that single message. If a receiver has multiple partially-received incoming messages from various senders, it sends grants only to the highest priority message among them; once it has granted all bytes of the highest priority message, it begins granting the next highest priority message, and so on.

The advantage of the single receiver assumption was that it relaxed the throughput constraint in our design. Under this assumption, the only contention point in the network is the downlink to the receiver from its TOR (top of the rack) switch. Therefore, our throughput goal was simplified to achieving high bandwidth utilization on that link alone. Furthermore, this simplified goal was trivially achieved through the receiver-driven flow control and packet scheduling we discussed in the previous two chapters.

As we relaxed the throughput constraint in previous chapters, we focused our design to achieve our primary goal: single-digit microseconds tail latency for short messages. By keeping one active message at a time at the receiver, the monogamous design minimizes the buffer occupancy and implements run-to-completion rather than round-robin scheduling. Both of these help to reduce message latencies. Furthermore, our low latency goal was achieved via implementation of the SRPT policy through a combination of receiver-driven packet scheduling and Homa's priority assignment scheme. We showed that this design can achieve near optimal tail latency for short messages while the receiver's downlink bandwidth is up to 80% utilized.

The issue with the monogamous design of Homa is that it cannot achieve high bandwidth utilization under many-to-many traffic patterns in datacenter networks. With realistic datacenter workloads, it's often the case that a large subset of servers in the network are transmitting flows to each other most of the time. So, not only there are many sender hosts in the network, but there are many receiver hosts in the network too. Unfortunately, the monogamous design of Homa from the previous chapters cannot utilize network bandwidth efficiently when we have a many-to-many traffic pattern. Our simulations showed that allowing only one active message with the monogamous design resulted in poor network utilization under high load. For example, in our simulations with workload W6 from Figure 4.2(a), Homa could not use more than 63% of the network bandwidth, regardless of how much load was offered by the senders.

The network remains underutilized because senders can not always respond immediately to grants when they receive them from multiple receivers. It happens very often that a sender wants to send messages to multiple receivers (e.g. N receivers). So the sender transmits unscheduled packets to all of them. Consequently, it receives grant packets from all of the receivers at the same time. But the sender is only able to transmit scheduled packets to one of the receivers at a time. This causes the downlinks for the other N-1 receivers to remain idle. Hence the downlink bandwidth for those receivers is wasted since no packets are received for the grants they sent. Figure 5.1 illustrates how this can happen.

We ran a set of simulation experiments with the topology of Figure 4.1 to measure the wasted bandwidth on the receiver downlinks. In a single experiment, each host acts as both sender and receiver. Each sender generates messages in an open loop fashion: the sender produces a series of messages that are sampled from one the workloads' message size CDF and each generated message is destined to a randomly chosen receiver. The interarrival time between messages follows a Poisson distribution, with the mean of the distribution set to achieve a certain offered network load on each sender's uplink. We then measured the wasted bandwidth on the receivers' downlinks as follows: a receiver wastes bandwidth when the downlink to the receiver is idle, but the receiver has more than one sender to schedule; i.e. the receiver doesn't receiver any packets on its downlink because it didn't schedule more than one sender. We measure the wasted bandwidth as the fraction of all total idle times for all receivers, divided the simulation times for all receivers.

Figure 5.2 depicts the wasted bandwidth for W6 under Homa's monogamous design. To compute this figure, we performed a set of simulations where each simulation was ran under a certain

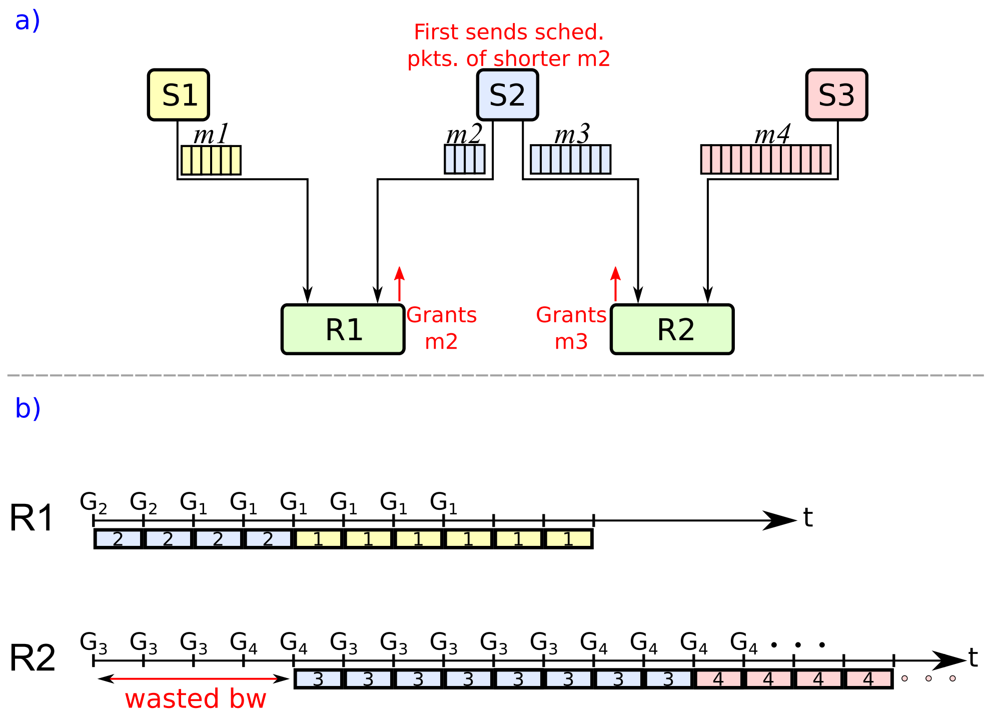


Figure 5.1: Bandwidth can be wasted if each receiver grants to a single sender at a time.
a) In this example, S1 has message *m1* for receiver R1, S2 has messages *m2* and *m3* for R1 and R2 respectively, and S3 has message *m4* for R2. In this scenario, R1 and R2 each grant to only one message at a time; R1 chooses *m2*, which is shorter than *m1*, and R2 chooses *m3*, which is shorter than *m4*. However, S1 can only send one packet at a time and it chooses to transmit *m2*, since it is shorter than *m1*. As a result, R2's downlink will be idle even though it could be used to receive *m4*.
b) The figure shows the time line of R1 and R2, starting when R2 sends the first grant packet for *m3*. RTTBytes is three packets in this example. At the top of the lines, the times at which grant packets are sent out are marked as G_i, with i for the message *m*i. Under each line, it shows the scheduled packets of the messages as they arrive at the receiver. Initially R1 and R2 both are granting *m2* and *m3* for the same sender S2. However, S2 only sends scheduled packets to R1. Consequently R2's downlink is idle, not receiving any packets for the grants it sent to S2 which leads to wasted bandwidth (the amount of time that R2's link remains idle is marked as "wasted bw" on the figure). Eventually R2 starts granting *m4* at the 4'th time tick, after it had granted *m3* for a full RTTBytes. But, by then it's already too late and R2 has wasted its bandwidth for one RTT.

network load. We increased the offered network load from 5% to higher loads and measured the wasted bandwidth on the receivers downlinks for each network load. The figure depicts the wasted bandwidth on the y-axis, as a function of the offered network load on the x axis. The black curve on the figure shows that as the offered load increases on the x axis, Homa's monogamous design wastes more and more bandwidth. At high network loads, the wasted bandwidth is significantly large under many-to-many traffic pattern for the heavy-tailed W6. That's because at higher loads there's a higher chance that the receiver sends a grant to a sender and the sender wont respond because it's busy transmitting to other receivers.

The slanted pink line on the figure shows the available surplus bandwidth, for each given offered network load on the x axis. For example, if the offered network load on the x axis is 25% of the total available bandwidth capacity, then the surplus bandwidth is 75% and wasted bandwidth on the y axis cannot exceed 75% on the pink line; i.e. the transport can't possibly waste more bandwidth than the total available surplus bandwidth. For a given network load, the transport may waste bandwidth at most up to the available surplus bandwidth.

From the figure, we can measured that the maximum achievable throughput for workload W6 with Homa's monogamous design is at most 63% of the host link bandwidth. The black curve on the figure shows that as the offered load increases on the x axis, Homa's monogamous design wastes higher bandwidth until the curve hits the pink surplus bandwidth line; wasted bandwidth increases as the network load increases and eventually it consumes all of the available surplus bandwidth. The offered load on the x axis at the intersection of the black curve and the pink line, defines the maximum achievable throughput with Homa's monogamous design under workload W6. If the offered load is higher than the maximum achievable throughput, then simulations becomes unstable and message queues grow without bound. Later in this chapter, we'll present the wasted bandwidth results for other workloads as well.

5.2 Overcommitment To Avoid Wasting Bandwidth

To avoid wasting bandwidth the receivers need to hedge their bet and grant to multiple receivers. The monogamous design wastes bandwidth because there is no way for a receiver to predict whether a particular sender will respond to grants or not. So the only way to keep the downlink highly utilized is to overcommit. Overcommitment means that a receiver must grant to more than one sender at a time, even though its downlink can only support one of the transmissions. With this approach, if one sender does not respond, then the downlink can be used for some other sender. However,

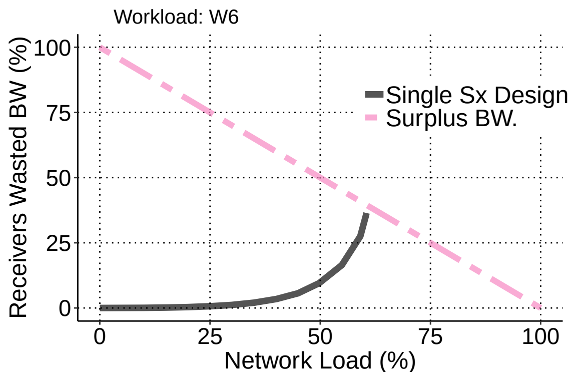


Figure 5.2: With Homa's monogamous design at most 63% of the receiver bandwidth can be utilized for workload W6 under many-to-many traffic pattern. The black curve show that as the offered load increases, Homa's monogamous design wastes more and more bandwidth. The offered load on the x axis at the intersection of the black curve and pink line, defines the maximum bandwidth that can be utilized with Homa's monogamous design.

overcommitment can also cause queuing at the TOR downlink to the receivers if multiple senders responds to the grants at the same time.

To ensure that buffer buildup (because of overcommitment) does not hurt the latency of the short messages, we utilize the priority mechanism in Homa, with shorter messages' packets at higher priorities. The priority mechanism ensures that the shortest message is delivered first and packets from the other messages will be buffered in the TOR. Figure 5.3 illustrates how the combination of the overcommitment and priority scheme of Homa achieves optimal performance: minimum latency for short messages along with maximum bandwidth utilization.

We use the term active to describe the messages for which the receiver is willing to send grants; the others are inactive. A receiver can stop transmission of a message and make it inactive by withholding grants; once all of the previously-granted data arrives, the sender will not transmit any more data for that message until the receiver starts sending grants again. Hence the message becomes inactive.

One of the important design decisions for Homa is how many active messages a receiver should allow at any given time. One possible approach is to keep all incoming messages active at all times. This is the approach used by TCP and most other existing protocols. However, if many senders respond at once this approach results in high buffer occupancy at the TOR near the receiver and round-robin scheduling between messages, both of which contribute to high tail latency.

We use the term degree of overcommitment to refer to the maximum number of messages that may be active (i.e. receive grants) at once on a given receiver. If there are more than this many scheduled messages expecting grants, only the highest priority ones are active. A higher degree of overcommitment reduces the likelihood of wasted bandwidth; with higher degree of overcommitment there's a higher chance that at least one of the active messages respond to grants. But it also leads to consuming more buffer space in the TOR (up to RTTbytes for each active message) when all active messages respond to grants at the same time. This results in more round-robin scheduling between messages, which increases average completion time.

Homa currently sets the degree of overcommitment to the number of scheduled priority levels: a receiver will grant to at most one message for each available scheduled priority level. This means that depending on the characteristics of the workload that the receiver measures, the degree of overcommitment will be different (in Section 3.5.1, we described how the receiver measures the workload of the incoming messages to allocate scheduled priority levels). Therefore, if for example a receiver determines that from a total of eight priority levels P_0 to P_7, the lower three priority levels P_0 to P_2 are to be used for scheduled packets, then the receiver sets the degree of overcommitment to three. It then grants the three shortest scheduled messages at these priority levels with the shortest message among them at better priority level P_2 and the longest one among them at P_0.

We chose to tie the degree of overcommitment to the number of scheduled priority levels because of the simplicity and effectiveness of this scheme in achieving both low latency and high throughput. Our approach to tie the degree of overcommitment to the available scheduled priority levels resulted in high network utilization in our simulations, but there are other plausible approaches. For example, a receiver might use a fixed degree of overcommitment, independent of available priority levels (if necessary, several messages could share the lowest priority level); or, it might adjust the degree of overcommitment dynamically based on sender response rates. We leave an exploration of these alternatives to future work. In the rest of this section we'll demonstrate and discuss the effectiveness of our scheme in more detail.

Figure 5.4 illustrates the effectiveness of the overcommitment mechanism in increasing network bandwidth utilization under different workloads. The figure shows the receivers' wasted bandwidth versus the offered load for six of the workloads we introduced in Figure 4.2(a). Each plot shows the simulation results as we increase the degree of overcommitment by increasing the number of available scheduled priority levels. For example, "1 Sched. Prio." curve on each plot corresponds to no overcommitment (i.e. single active scheduled message) and "2 Sched. Prio." corresponds to overcommitment degree of one (i.e. two active messages at a time). Regardless of the workload type, when the number of scheduled priorities increases, the wasted bandwidth decreases for the

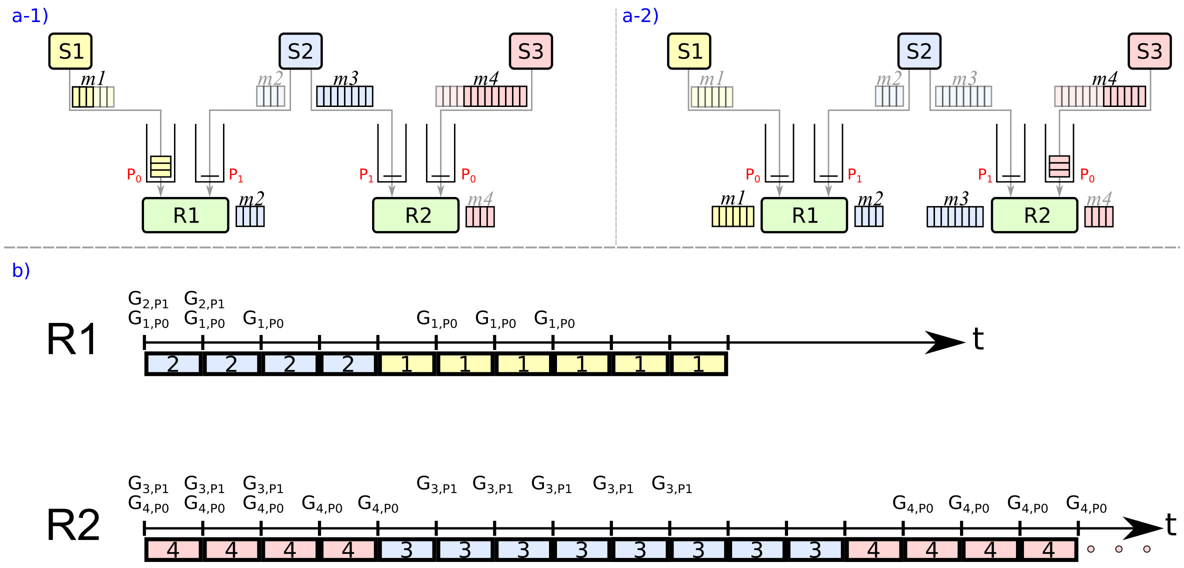


Figure 5.3: To achieve low latency and high bandwidth utilization, receivers overcommit by sending grants to multiple senders at the same time but at different priority levels. Shorter messages are granted at higher priority levels.
a) In the top left subfigure, R1 grants both m1 (from S1) and m2 (from S2). The higher priority packets from shorter message m2 are quickly delivered to R1 while lower priority packets from larger m1 will be buffered; this guarantees low latency delivery of shorter message m1. Meanwhile, R2 grants both m3 (from S2) and m4 (from S3). S2 transmits m2 not m3, but S3 transmits m4 and uses R2's downlink which ensures high bandwidth utilization at the R2's downlink. In the top right subfigure, S2 has finished transmitting m2 so it uses its grants to transmit m3's packets to R2 while S3 is also transmitting m4's packets to R2. This means the high priority packets from m3 will be delivered, while the lower priority packets from m4 will be buffered which again results in low latency delivery of the shorter message m3.
b) The time lines for R1 and R2 show how the overcommitment mechanism has avoided wasted bandwidth at R2's downlink (the downlinks to the receivers are always kept busy receiving data packets). On the top of each time line, grant packets are shown when they leave the granting receiver. Each grant is subscripted by the message number it's sent for and the scheduled priority level it carries for that message. Every time a new data packet arrives, it triggers the receiver to send a new grant for the messages. Each receiver sends enough grants to each message such that it keeps one RTTBytes (e.g. three packets in this example) of outstanding packets for each message.

same offered network load. This allows Homa to utilize more of the network bandwidth.

To achieve high bandwidth utilization, heavy-tailed workloads require higher degrees of overcommitment. Homa automatically assigns a degree of overcommitment that can achieve more than 90% network bandwidth utilization. For workload W1, only one scheduled priority is enough to achieve 90% network bandwidth utilization. However, for workloads W6 and W7, seven scheduled priorities are required to achieve close to 90% network bandwidth utilization. Network fabrics like

Ethernet typically provide eight priority levels. Homa dynamically assigns the appropriate number of scheduled priority levels based on the workload type to achieve 90% utilization of the network bandwidth. This dynamic assignment was explained in details in 3.5.1; in summary, each Homa receiver adaptively measures its incoming traffic and divide priorities between scheduled and unscheduled traffic to balance bytes between the two priority types. This mechanism would assign seven scheduled priorities for W6 and W7, one scheduled priority for W1, and so on for the rest of the workloads. Therefore it ensures up to 90% network bandwidth utilization for all workloads when eight priorities are provided by the fabric.

Here is how the overcommit mechanism works in Homa: each receiver dynamically assigns some number of priority levels for scheduled packets as we explained in Section 3.5.1. For instance, imagine that the receiver decided to allocate k priority levels (P_1 to P_k) for the scheduled packets. It then grants the $k'th$ shortest scheduled messages at these priority levels, with the shortest message among them at better priority level P_k and the longest one among them at P_1. Every time a new data packet arrives, it triggers the receiver to send a new grant for these messages at the correct priority level we just explained. The receiver sends enough grants to each message such that it keeps one RTTBytes of outstanding packets for each of these high priority messages. Every time a message is fully granted the receiver replaces it with the next shortest message in rank.

Unfortunately, Homa is unable to achieve 100% bandwidth because it still wastes network bandwidth under some conditions. Homa wastes bandwidth because it has a limited number of scheduled priority levels: there can be times when (a) all of the scheduled priority levels are allocated, (b) none of the senders that are granted is responding, so the receiver's downlink is idle and (c) there are additional messages for which the receiver could send grants if it had more priority levels. That said, we are not aware of any transport mechanism that can reach higher than 90% bandwith utilization on an arbitrary network topology and workload with an all to all random traffic matrix. In Chapter 6 we discuss the maximum bandwidth utilization of some of these transports.

The need for overcommitment provides illustrates why it isn't practical to completely eliminate buffering in a transport protocol; if we eliminate the buffers in the network, we run the risk of wasting the receivers' link bandwidths when the senders don't transmit their packets in a timely manner. In particular, Homa's monogamous design was our effort to eliminate buffering on the TOR downlinks but it turned out that the design could only support up to 63% link bandwidth utilization for heavy-tailed workloads. Homa introduces just enough buffering to ensure good link utilization; it then uses priorities to make sure that the buffering doesn't impact latency.

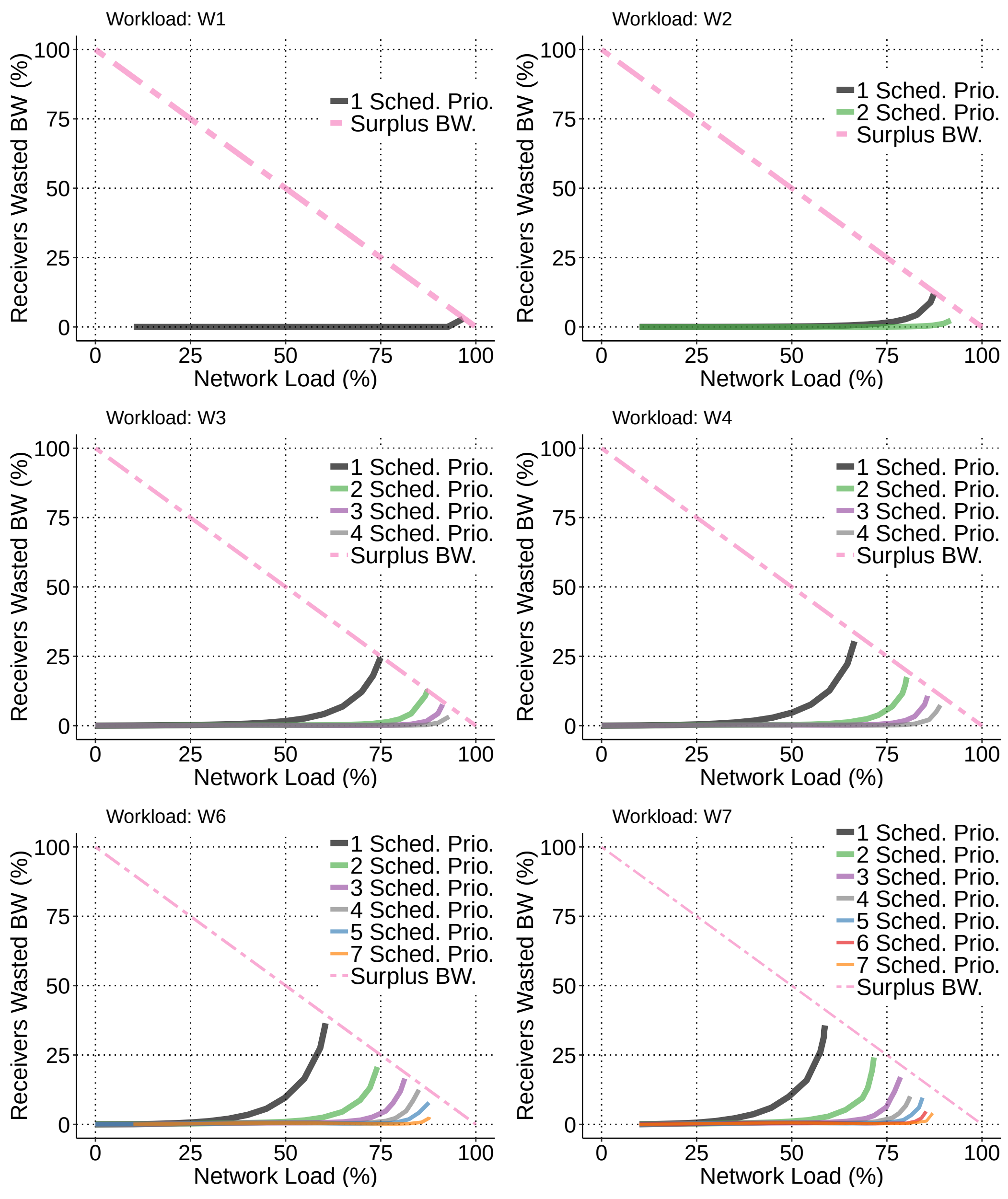


Figure 5.4: Increasing the degree of overcommitment increases the network bandwidth utilization for most of the workloads. For heavy-tailed workloads like W6 and W7, a higher degree of overcommitment is required to achieve 90% bandwidth utilization. Plots show the receiver's wasted bandwidth as a function of offered network load with different curves for different degrees of overcommitment. As we increase the number of scheduled priority levels, and hence the degree of overcommitment, less bandwidth is wasted for W2–W7. Wasting less bandwidth on the network translates to higher bandwidth utilization. This shows the importance of overcommitment: For example for W6, if receivers grant to only one message at a time, Homa can only support a network load of about 63%, versus 90% with an overcommitment level of 7. If the goal is to achieve close to 90% network bandwidth utilization, then workloads W1 and W2 require no overcommitment (i.e. one scheduled priorities), W3 requires overcommitment of two (i.e. three scheduled priorities), W4 requires overcommitment of three, and W6 and W7 require overcommitment of six. W5 is removed from the figure to keep the figure in one page (W5's results is very similar to W4's).

5.3 Chapter Summary

In this chapter we introduced the concept of overcommitment. We discussed the shortcomings of Homa's monogamous design in achieving high bandwidth utilization and we explained how the overcommit mechanism extends Homa's design to achieve high bandwidth utilization as well as low latency.

The overcommitment mechanism allows Homa to achieve up to 90% network bandwidth utilization for various workloads when the fabric provides eight priorities. Without overcommitment for heavy-tailed workloads, the bandwidth utilization on receivers' downlinks can be as low as 63%. In the current design of Homa, the degree of overcommitment is tied to the number of scheduled priorities. Homa receiver automatically assign enough scheduled priorities based on the measured workload such that high bandwidth utilization is achieved.

Chapter 6

Simulation Evaluation

This chapter is dedicated to the evaluation of the full design of Homa in a packet level simulator. The simulations allows us to explore more workloads, measure behavior at a deeper level, and compare with simulations of state of the art approaches like pFabric [5], pHost [14], NDP [16], and PIAS [7].

For Homa simulations, we used the packet-level simulator based on the OMNeT++ simulation framework [26]. The details of the simulator were discussed in §4.1 and we also added the over-commitment mechanism to the simulator. We measured pFabric, pHost, NDP, and PIAS using the original simulators developed by their authors. The pFabric simulator is based on ns-2 [25], and the PIAS simulator is based on the pFabric simulator. The pHost and NDP simulators were built from scratch without an underlying framework. We modified the simulators for pFabric, pHost, NDP, and PIAS to use the same workloads and network configuration as the Homa simulator. To the best of our abilities, we tuned each simulator to produce the best possible performance. The NDP simulator does not support less-than-full-size packets, so we used it only for workload W7, in which all packets are full-MTU-size datagrams.

Our simulations used an all-to-all communication pattern; each host was both a sender and a receiver, and the workload consisted of one-way messages transmitted from a sender to a receiver. New messages are created at senders according to a Poisson arrival process; the size of each message is chosen from one of the workloads W1 to W7 in Figure 4.2(a), and the destination for the message is chosen uniformly at random. For each simulation we selected a message arrival rate to produce a desired network load, which we define as the percentage of available network bandwidth consumed by goodput packets; this includes application-level data plus the minimum overhead (packet headers, inter-packet gaps, and control packets) required by the protocol; it does not include retransmitted packets.

The simulations do not model software queuing delays during packet processing in end-host stack. These kinds of software queuing delays occur when an incoming packet cannot be immediately processed because the receiver is still processing an earlier packet or the receiver is busy performing some other task unrelated to the network processing. Software delays are only modeled as fixed delays in the simulations for both transmit and receive paths of packets. The implications of fixed delays is that software can process packets with infinite throughput so no packet queuing happens in the software. The simulations only model the queuing effect in the network fabric.

Our goal with Homa simulations is to answer the following questions:

1. Does Homa provide low latency for short messages even at high network load and in the presence of long messages?
2. How efficiently does Homa use network bandwidth?
3. How does Homa compare to existing state-of-the-art approaches?
4. How important are Homa's novel features to its performance?

6.1 Comparison Transports

Before getting to the quantitative measurement of Homa's performance against prior designs, we discuss the details of these designs here.

pFabric

pFabric is a priority enabled transport mechanism that achieves near optimal latency for short messages while maintaining high network bandwidth utilization. pFabric was first to observe that smart usage of priority queues in the network fabric can significantly reduce flow completion times or message latencies. pFabric showed that it can achieve very low average and tail latency for various message sizes and its low latency performance is believed to be near-optimal. Therefore, it has been used as a comparison point for all of the low latency transport designs that came after. We also chose pFabric for comparison.

pFabric, like Homa, tries to approximate SRPT. The sender end-hosts, prior to transmission of each packet of a message, tag the packet with a priority level. The tagging takes place by writing the priority level in a designated field in the packet's header. The priority of the packet is set as the remaining size of the message that packet belongs to, at the transmission time of the packet. So for each packet the priority is computed independently.

The switches in the network are priority forwarding switches, giving forwarding preference to packets with lower priority tag. Each switch has a shallow output buffer per each output port. The packets in the buffer are sorted based on their priority tags and packets with lower priority levels are closer to the head of the queue. The switch forwards packets on the egress ports from the head of the queue. Because the value of the priority tag for each packet can range anywhere from 0 to the maximum message size in the workload (e.g. 100s of millions), the switches are required to support an infinite number of priority levels.

pFabric uses DCTCP [3] for the rate control mechanism. So, unlike Homa which uses a receiver side scheduler, pFabric uses a TCP-like sender-driven congestion control mechanism. A sender end-host initially transmits each message at the full rate that the NIC interface allows. The sender then reduces the transmission rate of the message, in reaction to Explicit Congestion Notification that network feeds back to it (following the DCTCP scheme).

Unlike Homa, pFabric requires modifications to switch hardware to support an infinite number of priority levels. pFabric makes heavy use of priority levels; in theory, the number of priority levels that pFabric requires is equal to the size of the largest message that is transmitted in the network. For example, if the largest message size in the workload is one megabyte or 1×10^6 bytes, then the number of distinct priority level that a packet can carry is 10^6. So network switches should support this many priority levels. This is a practical limitation of pFabric as this many priority levels are not available in commodity network fabrics. This means pFabric cannot be used in today's commodity network fabrics.

pHost

pHost, similar to Homa, uses a receiver-driven flow control mechanism and therefore, it's the closest related work to Homa. Under the pHost scheme, every time a new message arrives at a sender for transmission, the sender transmits an RTS (request-to-send) packet to the receiver for that message. The RTS packet carries information about size of the message, the sender address, etc. to the receiver. The receiver gathers this information from all messages pending for transmission and schedules them based on SRPT policy or any other policy of choice. The receiver schedules the messages by sending a token every MTU-sized packet time to one of the senders. Upon arrival of a token at the sender-host, the sender transmits one data packet from one its messages to the receiver that issued the token. Tokens in pHost play a similar role as grants in Homa. Furthermore, if a sender is presented with multiple messages to transmit at once, the sender is allowed to transmit RTS packets for all of the messages in parallel.

pHost introduces the concept of free tokens, which have the same functionality and goal as unscheduled packets in Homa; pHost tries to achieve low latency and high bandwidth utilization at the senders' links through free tokens per message. Each sender in pHost is granted one BDP (Bandwidth Delay Product, same as RTTbytes in Homa) worth of free tokens upon arrival of each message. The sender uses these free tokens to blindly transmit the first portion of the message. The combination of the free tokens and the receiver-driven scheduling leads to optimal message latency when there's a single receiver and single sender in an unloaded network.

Unlike Homa, pHost doesn't have any overcommitment mechanism; pHost tries to reduce the wasted bandwidth at the receivers' downlinks through a timeout mechanism on tokens. pHost argues that to avoid wasting bandwidth at receiver, the receiver has to detect nonresponsive senders and stop sending tokens to them and only send tokens to the responsive ones. To find the nonresponsive senders at the receiver, each token is augmented with a timeout value; upon the arrival of the token at the sender, the sender sets the timeout value to $1.5\times$ MTU-size packet transmission time. If the sender doesn't use the token within this time, the token is considered expired and sender won't be able to use it anymore. On the other end, the receiver tracks the number of expired tokens per sender (i.e. the difference between the number of tokens assigned and the number of packets received). If this number exceeds one BDP worth of tokens for a sender, then the receiver downgrades the sender by refusing to send any further tokens to the sender; the receiver instead sends tokens to the next sender in the pending message list. The receiver upgrades the sender again if it either receives a new packet from the sender or if the sender has remained downgraded for $3 \times BDP$.

The other difference between Homa and pHost is the usage of priorities; pHost only makes limited use of priority levels while Homa uses priorities extensively. pHost only uses two priority levels (high and low). Control packets like tokens and acknowledgment packets are transmitted at the high priority level. All other packets are transmitted at the low priority level. Additionally, pHost has the option to set high priority for the first BDP of packets of all messages as well.

Later in this chapter, we'll show how the differences between Homa and pHost lead to significant performance penalties for pHost; pHost cannot achieve optimal tail latency because of not using priorities extensively and is unable to maximize network bandwidth utilization because it wastes a lot of bandwidth.

NDP

NDP, similar to Homa and pHost, is a receiver-driven flow control mechanism. NDP makes the same observation as Homa that majority of the congestion in the network happens at the TOR's downlink to the receivers. When a new message arrives at an NDP sender, the sender transmits the first BDP bytes of the message at the full NIC transmit rate. This is crucial to achieve low latency for short messages. Beyond that first BDP bytes of the message, the sender wont send any more packets for that message until instructed by the receiver; that's to avoid additional congestion at TOR downlink to the receiver. The sender waits for arrival of pull packets (similar to tokens in pHost or grants in Homa) from the receiver that allow transmission of new packets at sender. On the other end, the receiver controls the rate of new packet arrivals by sending a new pull packet for every data packet it receives. In other words, the receiver paces the arrival of packets on its downlink to avoid congestion at the TOR switch.

Unlike Homa, lost packets are not rare in NDP; that's because NDP is designed to work with shallow buffers in the network that can frequently drop packets. To achieve low latency, NDP tries to minimize packet queuing in the network at all times; small buffers means less head of line blocking in the queues and therefore low latency for short messages. NDP's design sets the size of buffers to as small as eight packets per port. But the downside to small buffers is that packets can be routinely dropped because the queue fills up very quickly when multiple messages are transmitted at the same time to the same receiver.

Since packet loss is common in NDP, the key to achieving good latency is fast retransmission of dropped packets. But NDP observes that detecting lost packets and distinguishing between lost packets and late packets is a difficult task for the transport. This becomes even more difficult for NDP compared to other transport schemes, because lost packets are not rare in NDP. To detect lost packets quickly, NDP augments the switches with *cut payload* feature. With cut payload, when a packet arrives at a switch's queue and the queue is full, the switch doesn't drop the packet in its entirety; instead the switch cuts the payload of the packet and discards it and only keeps the header of the packet. The switch then priority-forwards the header of the trimmed packet, ahead of all data packets in the queue. The fast arrival of the priority-forwarded headers at the receiver allows the receiver to quickly detect the dropped packets. The receiver then requests a fast retransmission of the dropped packets by sending pull packets to the senders. This allows NDP to achieve good latency for messages even when packets are dropping frequently.

The main differences between Homa and NDP are: NDP doesn't use overcommitment, only makes limited use of priorities in the network, and uses fair-sharing scheduling policy instead of SRPT. NDP only uses two priority levels for its packets (one high and one low priority); the trimmed headers, pull packets, ack and nack packets are transmitted at high priority and all other packets are transmitted at low priority. Additionally NDP uses fair-sharing as the default scheduling policy at the receivers whereas Homa uses SRPT. We'll show later that because of these differences with Homa, NDP can't achieve optimal tail latency for short messages. Furthermore, because NDP doesn't have an overcommitment mechanisms, it waste significant amount of bandwidth and cannot achieve high bandwidth utilization. The last caveat with NDP is that it's not practical for today's commodity switches; it requires switch modifications to implement the cut payload feature.

PIAS

PIAS is an effort to approximate the optimal performance of pFabric without modifying anything in the network fabric or software stack. PIAS makes two observations about the limitations of today's networks and transports that makes pFabric infeasible. First, the network fabrics typically only provides a small number of priority levels, typically eight. Second, traditional POSIX compatible, Linux TCP sockets do not provide any interface where the size of a message is known at the time of transmission. To address these limitations, PIAS is designed such than unlike pFabric, it can be natively supported on today networks and it uses Linux DCTCP for congestion control.

To address these limitations of pFabric, PIAS tries to mimic Shortest Job First (SJF) scheduling, without having the priori knowledge of message sizes. PIAS leverages multiple priority queues available in existing commodity switches to implement a Multiple Level Feedback Queue (MLFQ), in which a message is gradually demoted from higher-priority queues to lower-priority queues based on the number of bytes it has sent. Since the message sizes are not known in advance, packets of each message are initially sent at the highest priority. As a message sends more and more packets, the priority of the packets are demoted to the lower priority levels. As a result, short flows are likely to be finished in the first few high-priority queues and thus be prioritized over long flows in general. This means that PIAS can be implemented without modifying the Linux TCP socket API and also PIAS can emulate SJF without knowing flow sizes beforehand.

PIAS tries to find priority demotion thresholds such that the average flow completion times (i.e. FCT) over all messages are minimized; it does this by deriving an analytical equation for the average FCT of all messages and optimizes the equation to find priority demotion thresholds. The priority demotion threshold between a lower priority P_i and a higher priority P_{i+1}, is a value in terms of bytes, such that if a message transmits more bytes than that value, the priority of its future packets is demoted to P_i. PIAS statically finds the demotion thresholds based on the workload message size distributions, similar to that of figure 4.2(a).

We chose to compare Homa against PIAS because it tries to achieve low latency goal by using priorities. However, PIAS has a few problems that prevent it from achieving low tail latency for short messages. First, it uses priorities in a more static fashion than Homa and it does not use receiver-driven scheduling. This causes large queues at high network loads and head of line blocking for short messages. Second, the MLFQ policy in PIAS behaves contrary to run-to-completion behavior; PIAS exhausts the high priority levels by over-transmitting on them (all messages regardless of their size start transmitting at high priority). This hurts the latency of short messages; the real high priority packets from short messages are blocked by packets of larger messages. Moreover, as a message sends more and more bytes, its priority decreases and its completion is further delayed. This makes it hard to finish longer messages and hurts their latency.

6.2 Homa's Latency vs. pFabric, pHost, PIAS, and NDP

Figures 6.1 and 6.2 display 99th percentile count spectrum plots (e.g. tail slowdown as a function of message size) at 80% and 50% network loads respectively. Each figure shows the results for workloads W1 to W7. Slowdown in each plot is measured in terms of one-way message delivery, from when the message is presented at the sender until it's fully delivered at the receiver. Unlike a typical plot where the x axis is linearly scaled with respect to the variable on the axis, the x axis on each plot on the figures is scaled to match the workload's message size CDF. As a result the x axis is linear in the number of messages in the workload. This type of plot was explained in detail in Section 4.3.1.

Most of the discussion below focuses on Figure 6.1(99th percentile at 80% network load) because it is the most challenging metric and the one that motivated Homa's design. The Homa curves in Figure 6.2 (99th percentile at 50% network load) are similar to those in Figure 6.1, but slowdowns are somewhat less in Figure 6.2.

SRPT-Oracle

Figure 6.1 compares Homa with SRPT-Oracle, an idealized "fluid" simulation of the SRPT scheduling policy. SRPT-Oracle represents the performance that would be achieved if (1) an optimal global SRPT matching between senders and receivers could be computed and disseminated to all servers instantaneously at all times; and (2) the network could realize the desired matching perfectly, for instance by preempting partially transmitted packets and balancing load perfectly. Every time a new message is presented to any of the senders or a message has finished transmission, SRPT-Oracle computes a set of distinct sender-receiver pairs such that the messages with shortest remaining bytes are transmitted to the receivers. SRPT-Oracle is not practical because it doesn't consider the impact

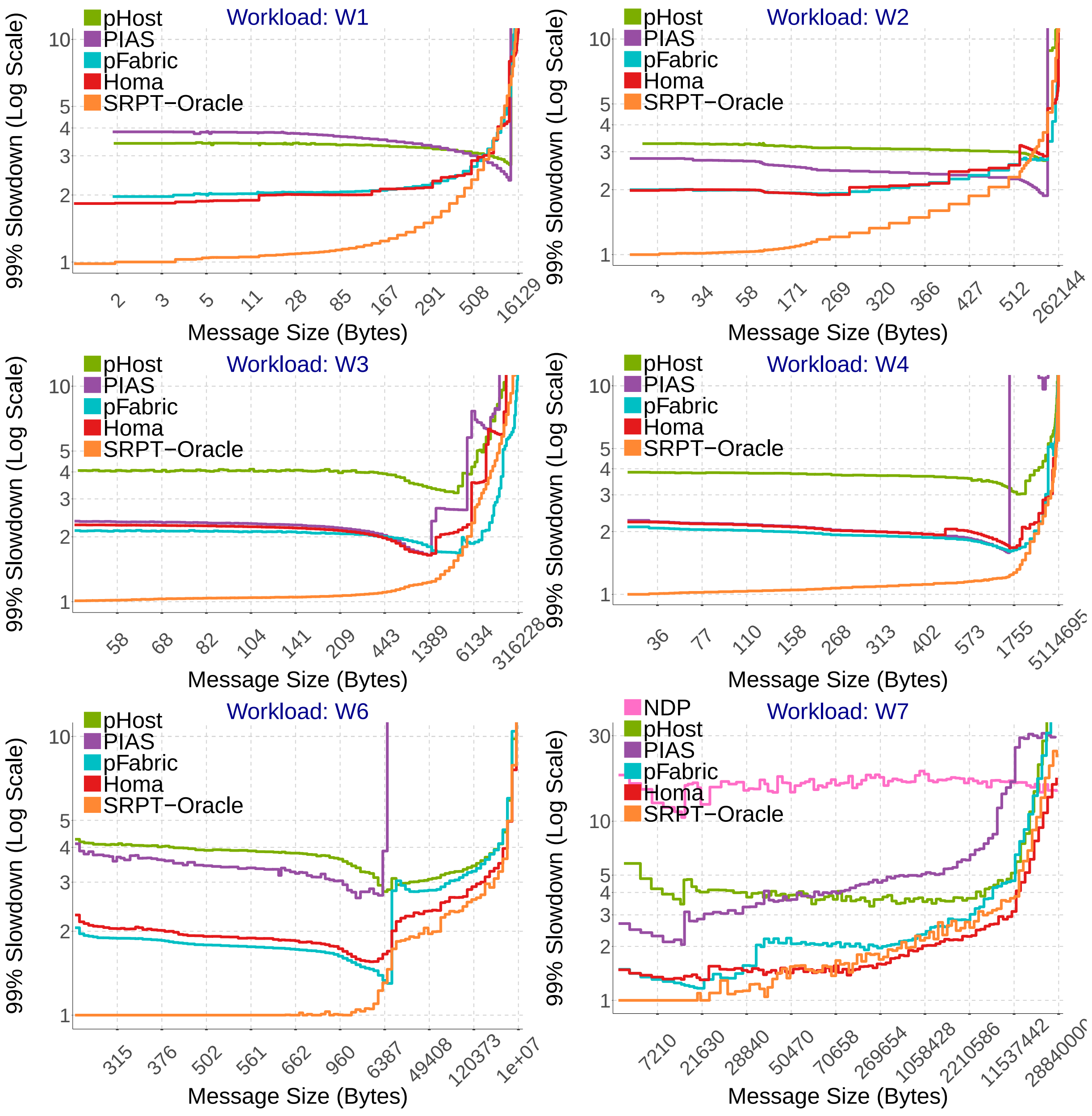


Figure 6.1: 99th-percentile slowdown as a function of message size, at 80% network load, for different protocols and workloads. Distance on the x-axis is linear in total number of messages (each tick corresponds to 10% of all messages). Graphs were measured at 80% network load, except for NDP and pHost. Neither NDP nor pHost can support 80% network load for these workloads, so we used the highest load that each protocol could support (70% for NDP, 58–73% for pHost, depending on workload). The minimum one-way time for a small message (slowdown is 1.0) is 2.3 µs. NDP was measured only for W7 because its simulator cannot handle partial packets. W5's plot is intentionally removed to keep all plots in one page (W5's plot is very similar to W6's plot in the figure).

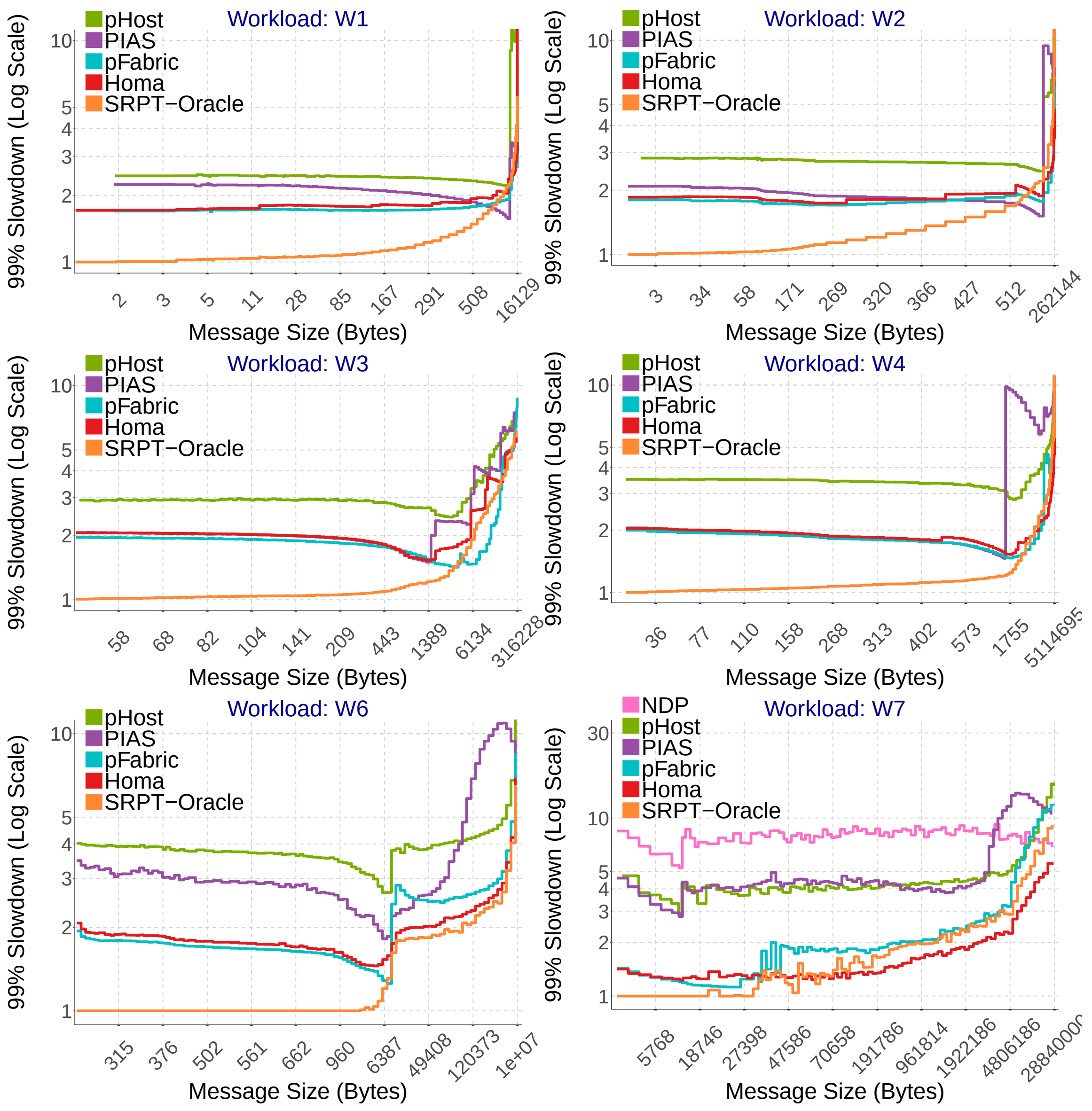


Figure 6.2: 99th-percentile slowdown as a function of message size, at 50% network load, for different protocols and workloads. Distance on the x-axis is linear in total number of messages (each tick corresponds to 10% of all messages). Graphs were measured at 50% network load. The minimum one-way time for a small message (slowdown is 1.0) is 2.3 µs. NDP was measured only for W7 because its simulator cannot handle partial packets.

of path collisions and bandwidth limitations in the network; it assumes there's an unblocked path in the network from every sender to every receiver for any packet that is scheduled to be transmitted. Also, it requires real time global knowledge and instantaneous communication of scheduling info. Although SRPT-Oracle is not practically realizable, it provides a reference for how well a protocol approximates the SRPT policy.

The results show that Homa achieves its design goal of mimicking SRPT scheduling; the form of Homa curves is similar to the SRPT-Oracle where tail slowdown for short messages is much less than long messages. The 99th percentile message slowdown for Homa is around 2$\times$ worse than SRPT-Oracle for tiny messages (e.g., $<$ 100 bytes). This translates to about 3 μs of extra latency for these messages, which we occurs because of partially transmitted packets that cannot be preempted in a real network. As the message size increases beyond RTTbytes (9.7 Kbytes), where Homa's packet scheduling mechanisms kick in, its latency curve closely matches the shape of SRPT-Oracle. pFabric is further away from SRPT-Oracle on workloads W6 and W7.

Homa has its worst tail slowdown performance for the largest messages of each workload at the far right of each plot. Generally this extremely bad tail performance happens for the largest 0.5% of messages in each workload. For example for W1 Homa leads to tail slowdown of larger than 11 for messages larger than 12-Kbytes and for W6 Homa leads to tail slowdown of 20 for messages around 10-Mbytes. This behavior is known as the starvation problem for very large messages with SRPT policy (as it is apparent from the Figure 6.1 the SRPT-Oracle has similar behavior for these extremely large messages).

One way to solve SRPT's starvation problem for very large messages is to allocate a small fraction (say 10%) of scheduling bandwidth for the messages that are starved; for example we can send one grant in every ten grants for messages which have been waiting for grants for a long time. That said, if we look at the mean slowdown plots of Figure 6.12, this issue is a lot less pronounced or it's even non existent. Therefore, one may argue that this starvation problem is not an issue worth resolving since mean slowdown is a much more relevant metric (than tail slowdown) for these extremely large and very rare messages in the workload.

pFabric, pHost, and PIAS

Homa delivers consistent low latency for small messages across all workloads, and its performance is similar to pFabric: 99th-percentile slowdown for the shortest 50% of messages is never worse than 2.2 at 80% network load. Homa's tail slowdown is at most 5% higher than pFabric's slowdown. For the large messages in heavy-tailed W6 and W7, Homa's slowdown is smaller than pFabric's slowdown. This is because of two reasons: 1) Homa maintains much shorter queues in the network comparing to pFabric therefore there's less head of line blocking for large messages. 2) Unlike Homa, packet drops are not rare in pFabric and larger messages experience more drops at the tail which increases tail slowdown for those messages.

pHost and PIAS have considerably higher slowdown than Homa and pFabric in Figure 6.1. This surprised us, because both pHost and PIAS claimed performance comparable to pFabric. On further review, we found that those claims were based on mean slowdown (in Figure 6.12 both pHost and PIAS provide performance closer to pFabric for mean slowdown). Our evaluation follows the original pFabric publication and focuses on 99th percentile slowdown.

PIAS and pHost have 1.5–4 times higher tails slowdown for short messages in heavy-head workloads W1-W4. The original pHost and PIAS papers didn't consider short messages in their evaluations. These papers evaluated their designs using only heavy-tailed workloads like W7 (workloads like W1 to W4 were not used in their evaluations) and all messages smaller than 100-Kbytes were considered short. In contrast, in workloads W1 to W4 majority of the messages are very short and fit in a single 1500-byte packet.

A comparison between the pHost and Homa curves in Figure 6.1 shows that a receiver-driven approach is not enough by itself to guarantee low latency. Both pHost and Homa are receiver-driven congestion control protocols. But, using priorities and overcommitment allows Homa to have near optimal tail latency. Because pHost is not using priorities and overcommitment, it has 2–3 times higher tail slowdown for short messages compared to Homa.

In our simulations for Figure 6.1, unfortunately pHost's design was not able to support 80% network load. When we tried to run the pHost simulations at 80% network load the simulator become unstable and the message queues grew without bound. So we compared against pHost at the highest network load that we could run an stable simulation for that workload; depending on the workload, pHost experiments were performed at 58%–73% network load. This means Homa compares even better against pHost than the figure suggests since the figure compares Homa at 80% network load against pHost at lower network loads.

The performance of PIAS in Figure 6.1 is somewhat erratic. Under most conditions, its tail latency is considerably worse than Homa, but for larger messages in W1 and W2 PIAS provides better latency than Homa. PIAS is nearly identical to Homa for small messages in workloads W3 and W4. PIAS always uses the highest priority level for messages that fit in a single packet, and this happens to match Homa's priority allocation for W3 and W4.

PIAS uses a multi-level feedback queue policy, where each message starts at high priority; the priority drops as the message is transmitted and PIAS learns more about its length. This policy is inferior to Homa's receiver-driven SRPT not only for small messages but also for most long ones. Small messages suffer because they get queued behind the high-priority prefixes of longer messages. Long messages suffer because their priority drops as they get closer to completion; this makes it hard to finish them. As a result, PIAS' slowdown jumps significantly for messages greater than one packet in length. In addition, without receiver-based scheduling, congestion led to ECN-induced backoff in workload W6, resulting in slowdowns of 20 or more for multi-packet messages. Homa uses the opposite approach from PIAS: the priority of a long message starts off low, but rises as the message gets closer to finishing; eventually the message runs to completion. In addition, Homa's rate-limiting and priority mechanisms work well together; for example, the rate limiter eliminates buffer overflow as a major consideration.

To show the advantage of SRPT, we made a trivial modification to PIAS. For short-message workloads such as W1, PIAS allocates multiple priority levels for the first packet worth of data. Rather than split the packet, PIAS transmits the entire packet at the highest priority level. We changed PIAS to use the lower of these priority levels, which makes it more SRPT-like. With this change, PIAS' performance became nearly identical to Homa's for messages less than one packet in length.

NDP.

The NDP simulator [16] could not simulate partial packets, so we measured NDP only with W7, in which all packets are full-size; Figure 6.1 shows the results. NDP's performance is considerably worse than any of the other protocols and we suspect that is due to two reasons. First, it uses a rate control mechanism with no overcommitment, which wastes bandwidth: at 70% network load, 27% of receiver bandwidth was wasted (the receiver had incomplete incoming messages yet its downlink was idle). The wasted downlink bandwidth results in additional queuing delays at high network load. We could not run the NDP simulation above 73% network load; at higher loads the simulation became unstable and message queues grew without bound.

NDP does not use SRPT and that's the second reason for why NDP has much higher tail slowdown; its receivers use a fair-share scheduling policy, which results in a uniformly high slowdown for all messages longer than RTTbytes. This demonstrates why SRPT is extremely important for achieving low latency; compared to Homa's SRPT, with NDP's fair-sharing the largest 1% of messages have better tail slowdown but the other 99% of messages are suffering. In addition, NDP senders do not prioritize their transmit queues; this results in severe head-of-line blocking for small messages when the transmit queue builds up during bursts. The NDP comparison demonstrates the importance of overcommitment and SRPT.

6.3 Bandwidth Utilization

To measure each protocol's ability to use network bandwidth efficiently, we simulated each workload-protocol combination at higher and higher network loads to identify the highest load the protocol can support (the load generator runs open-loop, so if the offered load exceeds the protocol's capacity, queues grow without bound). Figure 6.3 shows that Homa can operate at higher network loads than either pFabric, pHost, NDP, or PIAS, and its capacity is more stable across workloads. Homa's maximum bandwidth utilization ranges from 89%–92% depending on the workload while PIAS have bandwidth utilization in the range of 81%–85% and pFabric in the range of 75%–87%. NDP can utilize 73% of bandwidth for the one workload we simulated it. pHost is the worst of the protocols in terms of bandwidth utilization; pHost can only use 58%–73% of the network bandwidth depending on the workload.

None of the protocols can achieve 100% bandwidth because each of them wastes network bandwidth under some conditions. As we discussed earlier in this chapter, Homa wastes bandwidth because it has a limited number of scheduled priority levels: there can be times when (a) all of the scheduled priority levels are allocated, (b) none of those senders is responding, so the receiver's downlink is idle and (c) there are additional messages for which the receiver could send grants if it had more priority levels.

The other protocols also waste bandwidth. pFabric wastes bandwidth because it drops packets to signal congestion; those packets must be retransmitted later. NDP and pHost both waste bandwidth because they do not overcommit their downlinks. For example, in pHost, if a sender becomes nonresponsive, bandwidth on the receiver's downlink is wasted until the receiver times out and switches to a different sender. Moreover, in pHost the tokens expire in a small amount of time ($1.5 \times MTU$) and senders quickly miss the opportunity to send packets and can waste the receiver's downlink. Figure 6.3 suggests that Homa's overcommitment mechanism uses network bandwidth more efficiently than any of the other protocols.

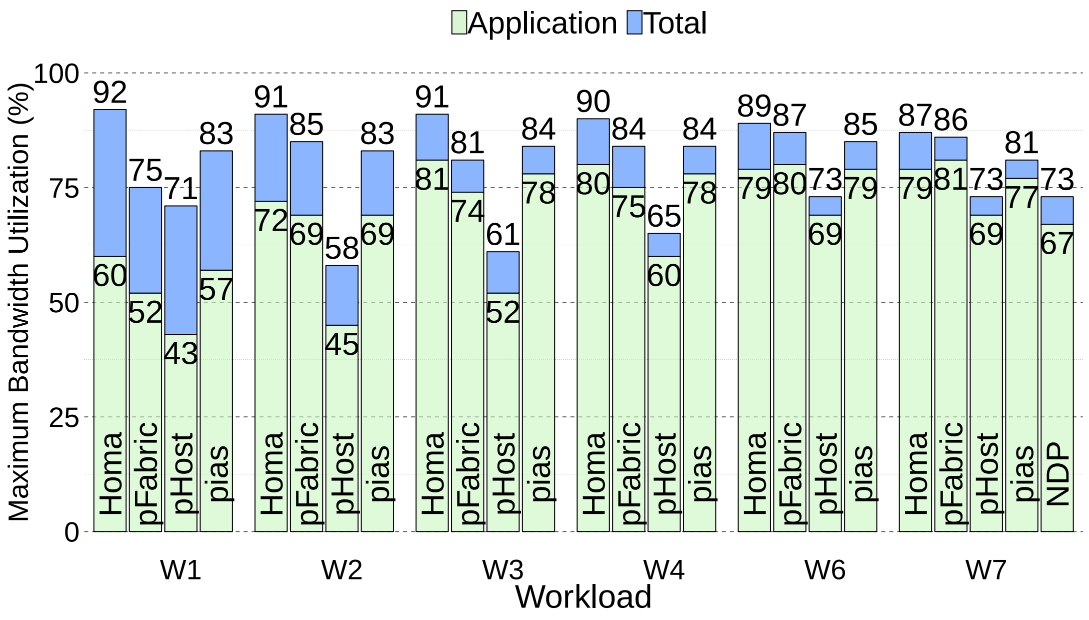


Figure 6.3: Network utilization limits. The bottom part of each bar indicates the percent of network bandwidth used for application data at that load. The top of each bar indicates the highest percent of available network bandwidth that the given protocol can support for the given workload; it counts all bytes in goodput packets. The top of the bars has higher value because it also accounts the overhead bytes in addition to the application data (including packet headers, and control packets; it excludes retransmitted packets).

6.4 Measuring Deeper Levels

In this section we present lower level measurements of the Homa protocol. These measurements help us to better understand the behavior of Homa's flow control and priority assignment mechanisms.

6.4.1 Causes of Remaining Delay

We instrumented the Homa simulator to identify the causes of tail latency ("why is the slowdown at the 99th percentile greater than 1.0?") Figure 6.4 shows that tail latency for short messages (i.e. the shortest 20% of messages in each workload) is almost entirely due to link-level preemption lag, where a packet from a short message arrives at a link while it is busy transmitting a packet from a longer message; when this happens transmission of the packet from short message is delayed until the current packet is fully serialized into the link. This shows that Homa is nearly optimal given the current networking hardware: the only way to significantly improve short messages' tail latency is

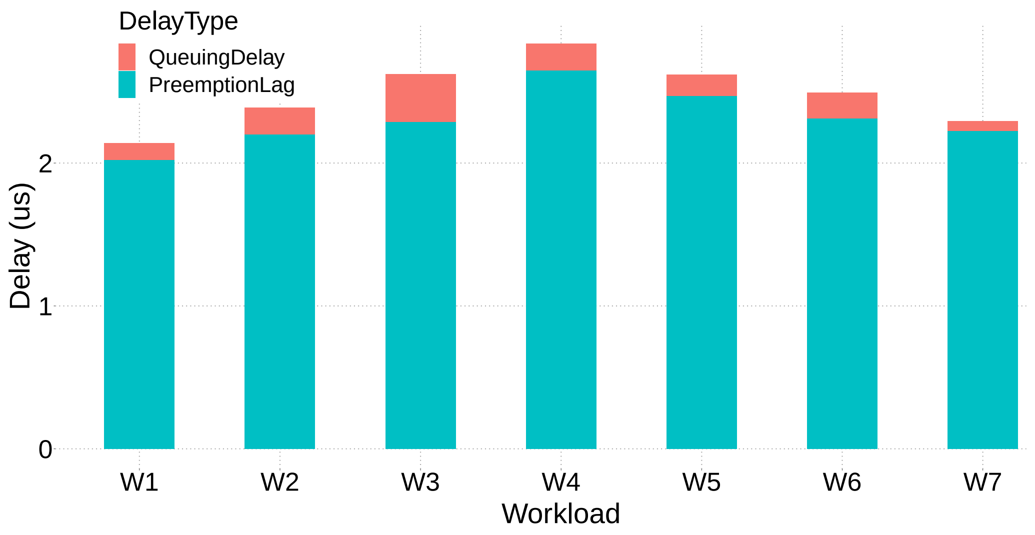


Figure 6.4: Sources of tail delay for short messages. “Preemption Lag” occurs when a higher priority packet must wait for a lower priority packet to finish transmission on a link. “Queuing Delay” occurs when a packet waits for one or more packets of equal or higher priority. Each bar represents an average across short messages with delay near the 99th percentile. For workloads W1-W6 the bar considers the smallest 20% of all messages; for W7 it considers all single packet messages.

with changes to the networking hardware, such as implementing link-level packet preemption.

6.4.2 Queue Length

Some queuing of packets in switches is inevitable in Homa because of its use of unscheduled packets and overcommitment. Even so, Table 6.1 shows that Homa is successful at limiting packet buffering: average queue lengths at 80% load are only 1–18 Kbytes, and the maximum observed queue length was 202 Kbytes (in a TOR→host downlink). Of the maximum, overcommitment accounts for as much as 56 Kbytes (RTTbytes in each of 7 scheduled priority levels); the remainder is from collisions of unscheduled packets. Workloads with shorter messages consume less buffer space than those with longer messages. For example, the W1 workload uses only one scheduled priority level, so it cannot overcommit; in addition, its messages are shorter, so more of them must collide simultaneously in order to build up long queues at the TOR. The 202-Kbyte peak occupancy is well within the capacity of typical switches, so the data validates our assumption that packet drops due to buffer overflows will be rare.

Table 6.1 also validates our original hypothesis that in datacenter networks the majority of congestions happens in the TOR→host downlinks and there is not significant congestion in the core. In

Queue		W1	W2	W3	W4	W5	W6	W7
TOR→Aggr	mean	0.7	1.0	1.7	1.6	1.7	1.7	1.7
	max	21.1	30.0	50.1	50.3	83.8	82.7	93.6
Aggr→TOR	mean	0.8	1.1	2.2	1.8	1.9	1.7	1.6
	max	22.4	34.1	78.7	57.1	84.6	92.2	78.1
TOR→host	mean	1.7	5.5	15.4	12.8	18.1	17.3	17.3
	max	58.7	93.0	202	117.9	148.2	146.1	126.4

Table 6.1: Average and maximum queue lengths (in Kbytes) at switch egress ports for each of the three levels of the network, measured at 80% network load. Queue lengths do not include partially-transmitted or partially-received packets.

Queue		W1	W2	W3	W4	W5	W6	W7
TOR→Aggr	mean	0.7	1.0	1.7	1.6	2.3	1.8	1.6
	max	16.5	27.5	50.3	55.1	69.8	84.7	115.2
Aggr→TOR	mean	0.8	1.1	2.2	2.0	3.1	1.9	1.5
	max	21.9	36.5	86.6	75.6	103.6	79.8	69.2
TOR→host	mean	1.7	6.5	26.5	23.2	45.0	37.2	34.3
	max	63.1	126.5	299.2	202.3	266.5	278.7	223.3

Table 6.2: Same as Table 6.1 except the value of RTTBytes in these experiments is two times larger than the one for Table 6.1.

spite of Homa's attempt to control the queues at the TOR→host downlinks, these queues can have mean occupancies up to 18 Kbytes and their maximum can reach as high as 202 Kbytes. The table confirms that the core queues (i.e. TOR→Aggr and Aggr→TOR) contain less than 2 Kbytes of data on average, and their maximum length is less than 100 Kbytes.

Homa appears to produce even lower buffer occupancy than Fastpass [28], which uses a centralized message scheduler to control queue lengths. Direct comparisons with Fastpass must be taken with a grain of salt, due to differences in experimental setup, but Fastpass produced a mean queue length of 18 Kbytes and a 99.9th percentile length of 350 Kbytes.

One question we like to answer is how sensitive are the queue occupancies, with respect to the changes in the value of RTTBytes? This is an interesting question because sometimes it's very challenging to measure RTTBytes accurately in the network and also different host-pairs in the network may have different RTTBytes (e.g. nodes within a rack have a different RTT value from nodes across two racks). In these situations we'd rather set RTTBytes to a higher value than a lower value (refer to Figure 6.10 for a discussion on this topic). So in these cases it is important to know

Queue		**W1**	**W2**	**W3**	**W4**	**W5**	**W6**	**W7**
TOR→Aggr	mean	0.7	1.6	1.7	1.6	2.4	1.7	1.6
	max	18.1	22.0	52.1	53.1	88.0	87.9	147.4
Aggr→TOR	mean	0.7	1.7	2.8	2.3	4.3	2.1	1.5
	max	17.7	47.1	122.1	144.3	279.6	137.2	122.4
TOR→host	mean	1.7	15.2	43.6	55.3	119.6	86.1	81.6
	max	76.3	321.1	608.8	538.5	805.9	700.3	577.6

Table 6.3: Same as Table 6.1 except the value of RTTBytes in these experiments is five times larger than the one for Table 6.1.

that if we use a larger value for RTTBytes, how does that impact the queue occupancies in the network? We need to make sure that queues don't grow too large and still remain within the buffer space provided by the switches.

Our simulations show that increasing the value of RTTBytes by a factor of two doesn't significantly increase the average and maximum queue sizes under Homa. Tables 6.2 and 6.3 show the queue lengths when we multiply the value of RTTBytes in our experiments by a factor of two and five respectively. Doubling RTTBytes has increased the average queue length to as high as 45 Kbytes and the maximum to 299 Kbytes. Both of these queue values are still within the buffer capacity of the commodity switches. These results are encouraging as they show that our computation of RTTBytes in Homa can have a large margin of error without incurring any issues for the transport; Homa may use an RTTBytes value that deviates by 100% from the correct value and yet packet drops because of buffer overflow remain rare in the network. Increasing the RTTBytes by factor of five results in excessively large queue sizes; the maximum queue occupancy in this case is around 806 Kbytes and Homa may start experiencing packet drops under high network load in this case.

6.4.3 Senders' SRPT

As we discussed in Section 3.6, Homa senders implement SRPT for their outgoing messages. In fact as we were designing Homa, we came to the understanding that an efficient implementation of SRPT at the senders is critical for achieving great tail slowdown for short messages. To implement senders' SRPT, Homa senders don't consider the priorities in the packets since these priorities are set to achieve SRPT at the downlinks to the receivers; the sender's priority for an outgoing packet does not necessarily correspond to the priority stored in the packet.

One of the most important and subtle design features of Homa that enables SRPT at the senders

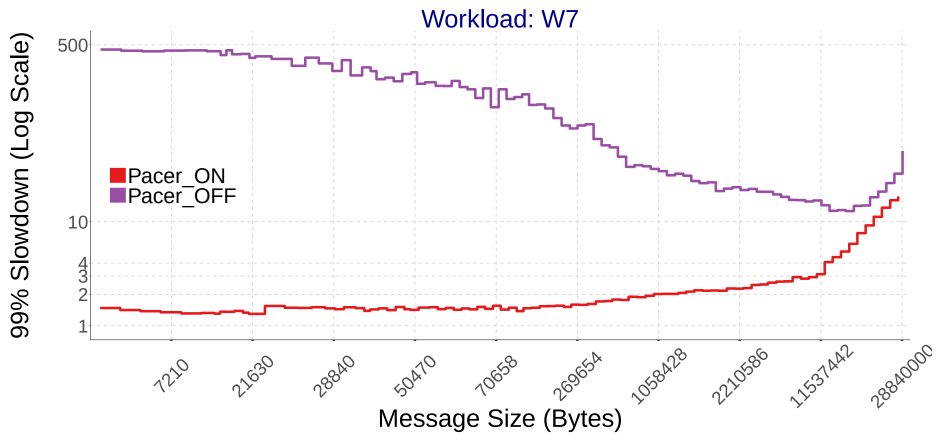


Figure 6.5: Impact of enabling/disabling senders' traffic pacers on the tail slowdown of workload W7 at 80% network load. When the transmit pacer is off, a queue builds up in the NIC's transmit queue and the sender fails to enforce SRPT among its outgoing messages. This increases the tail slowdown by more than two orders of magnitude for short messages.

is a traffic pacer that we implemented on the transmit path for the outgoing packets. Disabling this pacer in Homa increases the tail slowdown for short messages by orders of magnitude. Figure 6.5 shows the impact of disabling senders' traffic pacers on the tail slowdown of workload W7 at 80% network load. The crucial role of this pacer is that it prevents large queue build up in the NIC's transmit queue (i.e. ensures that the queue size always remains less than two packets) so that high priority outgoing packets don't have to wait for lower priority packets queued previously in the NIC queue. When this pacer is disabled, queue build up in the NIC prevents the sender from enforcing SRPT among the packets of its outgoing messages. This increases the tail slowdown by more than two orders of magnitude for the short message in W7. For other workloads, when pacer is disabled the tail slowdown of short messages increases by 8–20X.

6.4.4 Priority Utilization

Figure 6.6 shows how network traffic is divided among the priority levels when executing workload W4 at three different network loads. For this workload Homa splits the priorities evenly between scheduled and unscheduled packets. The four unscheduled priorities (P_4–P_7) are used evenly, with the same number of network bytes transmitted under each priority level. As the network load increases, the additional traffic is also split evenly across the unscheduled priority levels.

The four scheduled priorities are used in different ways depending on the network load. At 50%

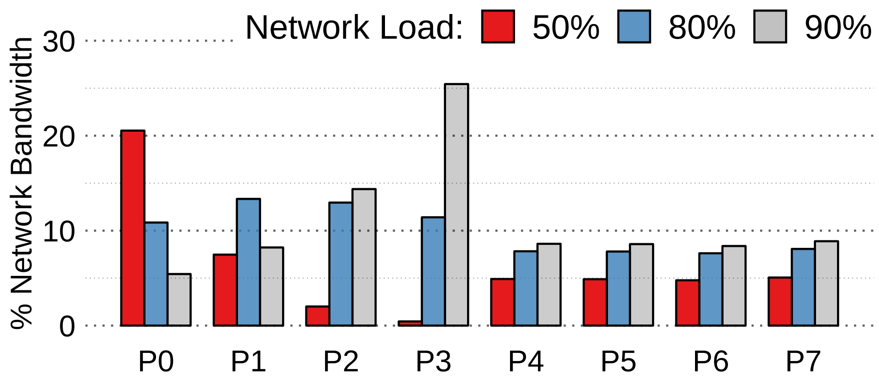


Figure 6.6: Usage of priority levels for workload W4 under different loads. Each bar indicates the number of network bytes transmitted at a given priority level, as a fraction of total available network bandwidth. P0-P3 are used for scheduled packets and P4-P7 for unscheduled packets.

load, a receiver typically has only one schedulable message at a time, in which case the message uses the lowest priority level (P0) and most of the scheduled traffic is on P0. Higher priority levels are used for preemption when a shorter message appears part-way through the reception of a longer one. It is rare for preemptions to nest deeply enough to use all four scheduled levels. As the network load increases, the usage of scheduled priorities changes. By the time network load reaches 90%, receivers typically have at least four partially-received messages at any given time, so they use all of the scheduled priority levels. More scheduled packets arrive on the highest scheduled level than any other; the other levels are used only if the highest-priority sender is nonresponsive or if the number of incoming messages drops below 4. The figure indicates that senders are frequently nonresponsive at 80% network load (more than half of the scheduled traffic arrives on P0–P2).

6.4.5 Configuration Policies

In Chapter 3 we presented Homa's priority allocation scheme for unscheduled packets and between unscheduled and scheduled packets. Then in Chapter 4 we evaluated this scheme under network load pressure when there is a single receiver in the network. Here we empirically evaluate the configuration scheme under many-to-many traffic patterns to show that our design of priority allocation design remains near optimal even when we have such traffic patterns.

Homa automatically configures itself to handle different workloads. For example, it allocates seven priority levels for unscheduled packets in W1, four in W4, and only one in W6 and W7. In

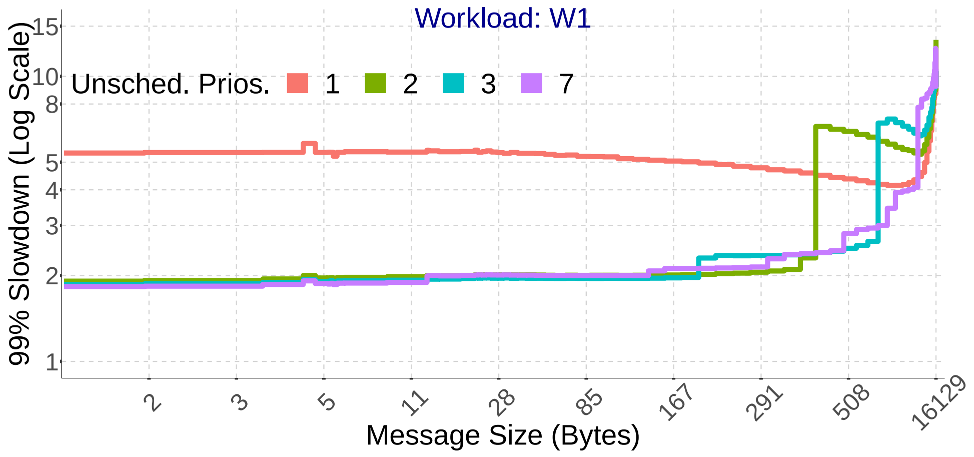


Figure 6.7: Impact of the number of unscheduled priority levels on workload W1 (80% network load, one scheduled priority level). The vertical jumps occur at the cutoff points between priority levels; for example with two priority levels Homa sets the cutoff point at around 350-byte.

this section we evaluate Homa's configuration policies by manually varying each parameter in order to see its impact on the performance. For each policy we display results for the workload with the greatest sensitivity to the parameter in question. The results show that Homa policies result in near optimal values for the configuration parameters.

One policy question to ask is "**Does Homa allocate the right number of priorities for unscheduled packets?**" To answer this questions we selected workload W1 and varied the number of unscheduled priorities for it. Figure 6.7 shows the slowdown for workload W1 when the number of unscheduled priorities was varied from 1 to 7 while fixing the number of scheduled priorities at 1 (Homa would normally allocate 7 unscheduled priorities for this workload). The graph shows that workloads with small messages need multiple unscheduled priorities in order to provide low latency: with only a single unscheduled priority (two priority levels in total), the 99th percentile slowdown increases by more than 2.5x for most message sizes. A second unscheduled priority level improves latency for more than 80% of messages; additional priority levels provide smaller gains but reduce the slowdown for the rest of the messages (except the largest of them). This figure suggests that for good slowdown performance Homa doesn't necessarily require eight priority levels; but three or four priority levels should be sufficient. However, if total of eight priority levels are provided, then Homa uses them for the unscheduled packet in this workload to get a closer approximation of the optimal SRPT. Note that Homa's performance is superior to that of NDP and pHost as both of these protocols only use a single priority level for all data packets.

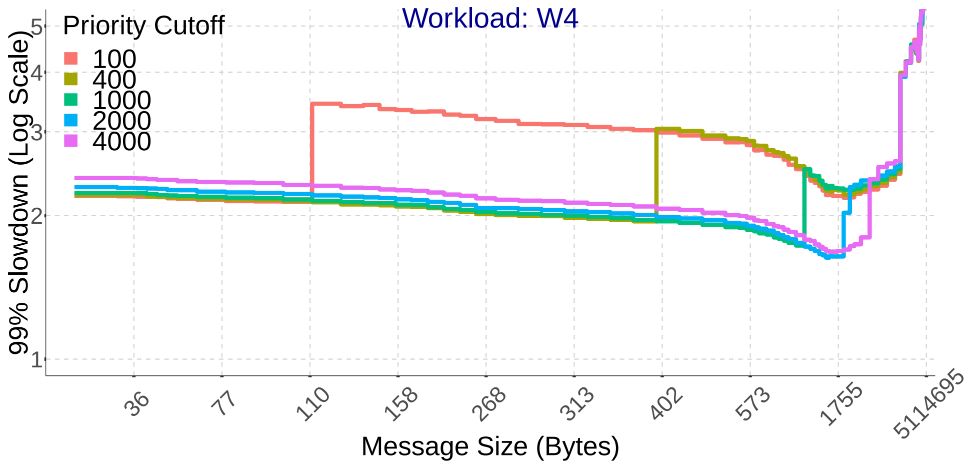


Figure 6.8: The impact of the cutoff point between unscheduled priorities for workload W4. All measurements were taken at 80% network load with 2 unscheduled priority levels. Each curve uses a different cutoff point between the two unscheduled levels. Homa's algorithm for choosing the cutoff, which balances the amount of network traffic on the two levels, would select a cutoff point of 1930 bytes.

Next we would like to evaluate "**How efficient is Homa in dividing the unscheduled priority levels between different message sizes?**". Figure 6.8 analyzes Homa's policy for choosing the cutoff points between unscheduled priority levels. It shows the 99th percentile slowdown for workload W4 when two priority levels are used for unscheduled packets and the cutoff point is varied. Increasing the cutoff point provides a significant latency reduction for messages sizes between the old and new values of the cutoff, while increasing latency slightly for smaller messages. Up until about 2000 bytes, the penalty for smaller messages is negligible; however, increasing the cutoff to 4000 bytes results in a noticeable penalty for about 90% of all messages, while providing a large benefit for about 5% of messages. A cutoff of around 2000 bytes appears to provide a reasonable balance. Homa's policy of balancing traffic in the levels would choose a cutoff point of 1930 bytes. We considered other ways of choosing the cutoffs, such as balancing the number of messages across priority levels; in Figure 6.8 this would place the cutoff around 200 bytes, which is clearly sub-optimal.

Next we would like to evaluate "**how does varying the number of scheduled priorities affect the slowdown for heavy-tailed workloads?**" We focus on heavy-tailed workloads here because they are the most sensitive to the number of scheduled priorities. Figure 6.9 shows the slowdown for workload W6 with four or seven scheduled priorities, while fixing the number of unscheduled priorities at 1 (Homa would normally allocate 7 scheduled priorities for this workload). Additional scheduled priorities beyond four have little impact on latency. However, the additional scheduled

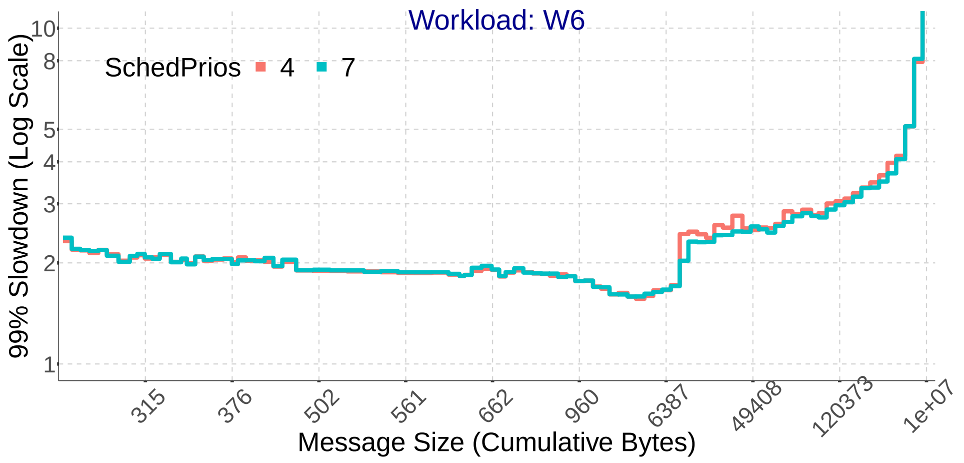


Figure 6.9: Impact of the number of scheduled priority levels on workload W6 (80% network load, one unscheduled priority level). Increasing the number of scheduled priorities beyond four has little to no effect on the tail slowdown.

priorities have a significant impact on the network load that can be sustained (refer to the overcommitment discussion in Chater 5 and in Figure 5.4). We do not need many priority levels to achieve SRPT, but we need them for high bandwidth utilization; this workload could not run at 80% network load with fewer than four scheduled priorities. In this figure we don't show the results for less than scheduled four priorities because Homa needs at least four scheduled priorities to achieve 80% bandwidth utilization.

The next feature of Homa we would like evaluate is "**the impact of the number of unscheduled bytes on the tail slowdown.**" Figure 6.10 shows the slowdown for workload W6 when the number of unscheduled bytes per message is varied. In this experiment, Homa uses one priority level for unscheduled packets and seven for the scheduled packets. The figure demonstrates the benefits of unscheduled packets: messages smaller than RTTbytes but larger than the unscheduled limit suffer 2.5x worse latency. Increasing the unscheduled limit beyond RTTbytes results in slightly worse performance for messages smaller than RTTbytes, because of additional traffic sharing the single unscheduled priority level. From this figure we conclude that in terms of latency performance, it is better to set RTTbytes at higher values than lower ones; higher values of RTTbytes help to reduce the latency but result in additional buffering as we discussed previously in this chapter.

The last question we would like to answer is "**how effective is Homa's policy in dividing the priority levels between unscheduled and scheduled traffic?**" Homa divides priorities between scheduled and unscheduled packets in proportion to the total number of bytes arriving in each kind

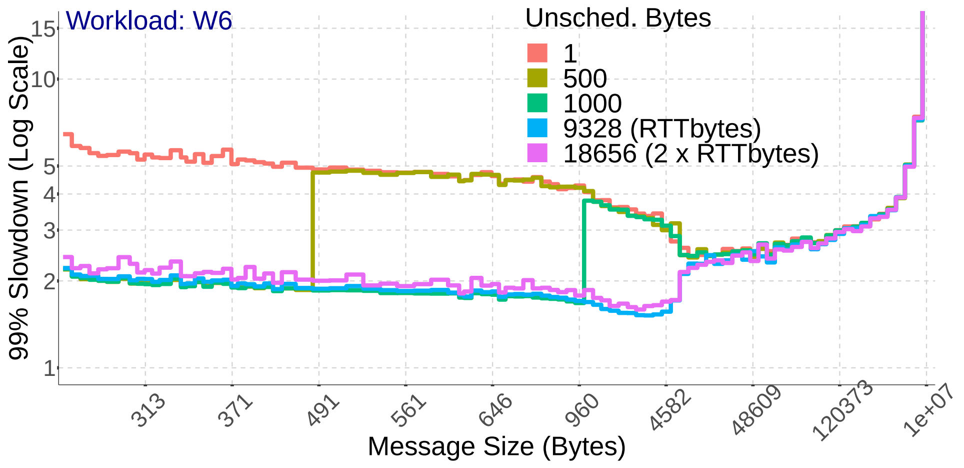


Figure 6.10: The impact of the number of unscheduled bytes on slowdown. Each curve uses a different limit on the number of unscheduled bytes per message. All measurements used workload W6 at 80% network load. The figure shows that it is important to send at least RTTBytes of unscheduled bytes per message for good tail slowdown across the spectrum. Sending more than RTTbytes of unscheduled bytes doesn't have any benefit for slowdown but at the same time it is better to set RTTbytes at higher values rather than lower ones.

of packet and balances traffic bytes between the two priority type. To evaluate this choice, we simulated workload W3 under different allocations between scheduled and unscheduled priorities. We chose W3 because it is most balanced between scheduled and unscheduled packets; Homa will normally allocate four priorities for each. The results are shown in Figure 6.11. If Homa shifts the balance in the direction of more scheduled priorities, the tail latency for short messages starts to increase; with seven scheduled priorities the short messages' slowdown is 2.5x the minimum slowdown. If Homa shifts the balance in the other direction (fewer scheduled priorities), there is relatively little impact on latency. This shows Homa's choice of four scheduled and four unscheduled priorities is the right priority allocation for this workload.

6.5 Mean and Median Latency: Homa vs. Other Protocols

For the curious readers, in this section we added the mean and median slowdown spectrum plots for Homa vs other protocols. Figures 6.12 and 6.13 displays mean slowdown for all workloads and all protocols. And, Figures 6.14 and 6.15 displays median slowdown for the same experiments. As we discussed earlier, even on these slowdown metrics, Homa outperforms PIAS, pHost and NDP. However, the difference between the performance of Homa and other protocols is less pronounced.

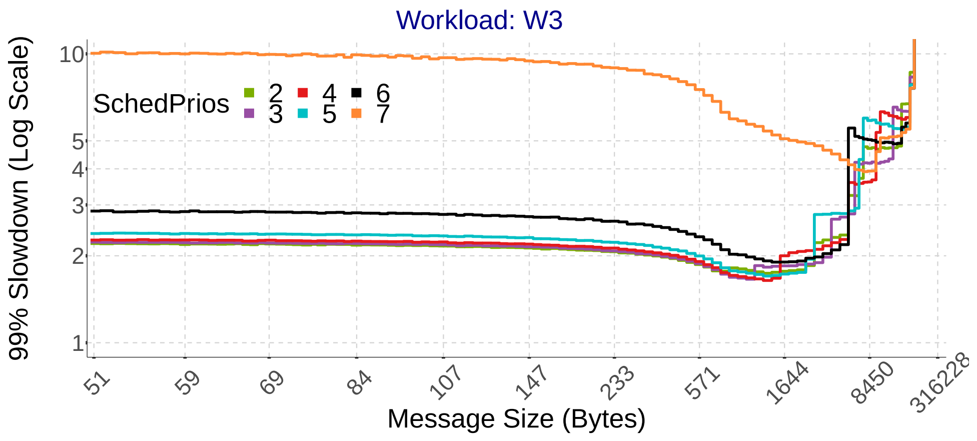


Figure 6.11: The impact on slowdown of the balance between unscheduled and scheduled priority levels. Each curve uses 8 total priority levels, but with a different trade off between unscheduled and scheduled levels. All measurements used workload W2 at 80% network load.

6.6 Chapter Summary

In this chapter we evaluated the end-to-end design of Homa using a packet simulator that we developed in OMNeT++ framework. We showed that Homa indeed can achieve its goals of low latency and high throughput. We showed that for a variety of of workloads, Homa can achieve 99%-ile tail slowdown of less than 2.2 for the shortest 50% of all messages in each workload. For example, 99%-ile one-way latency for a 100-byte message is 4–5µs in a two tier network topology. Additionally we demonstrated that Homa can achieve up to 90% network bandwidth utilization for a variety of workloads.

We also compared Homa against pFabric, pHost, PIAS and NDP. Homa achieves very similar tail latency to pFabric, without the switch modifications that pFabric needs. 99%-ile tail slowdown at 80% network load under PIAS and pHost is at least 1.5–4x larger than that of Homa. Homa also outperforms the other protocols in terms of bandwidth utilization.

Lower level measurements of Homa shows that Homa is close to optimal in terms of it priority allocation mechanisms. When we look at the breakdown of the tail latency for short messages under Homa, we observe that the majority of the extra latency is because of preemption lag in the switches when a short packet is ready to be forwarded at an egress port but it has to be delayed for a larger packet currently being transmitted on the port. This preempting lag is a limitation of the switch fabric. In order to further improve the tail latency for the short messages, switches need to add the support of preempting packets in the middle of transmission.

We also showed that as much as the receivers' SRPT is important for low latency, so is the senders' SRPT. Homa senders can enforce SRPT using a packet pacer that controls queue build up in the NIC's transmit queue. Disabling this pacer prevents the senders to correctly implement SRPT among outbound messages which increases the tail slowdown for short messages by two orders of magnitude.

Finally, we also evaluated Homa's configuration policy and showed that Homa does a good job of setting various parameters to achieve very low tail latency in various situations.

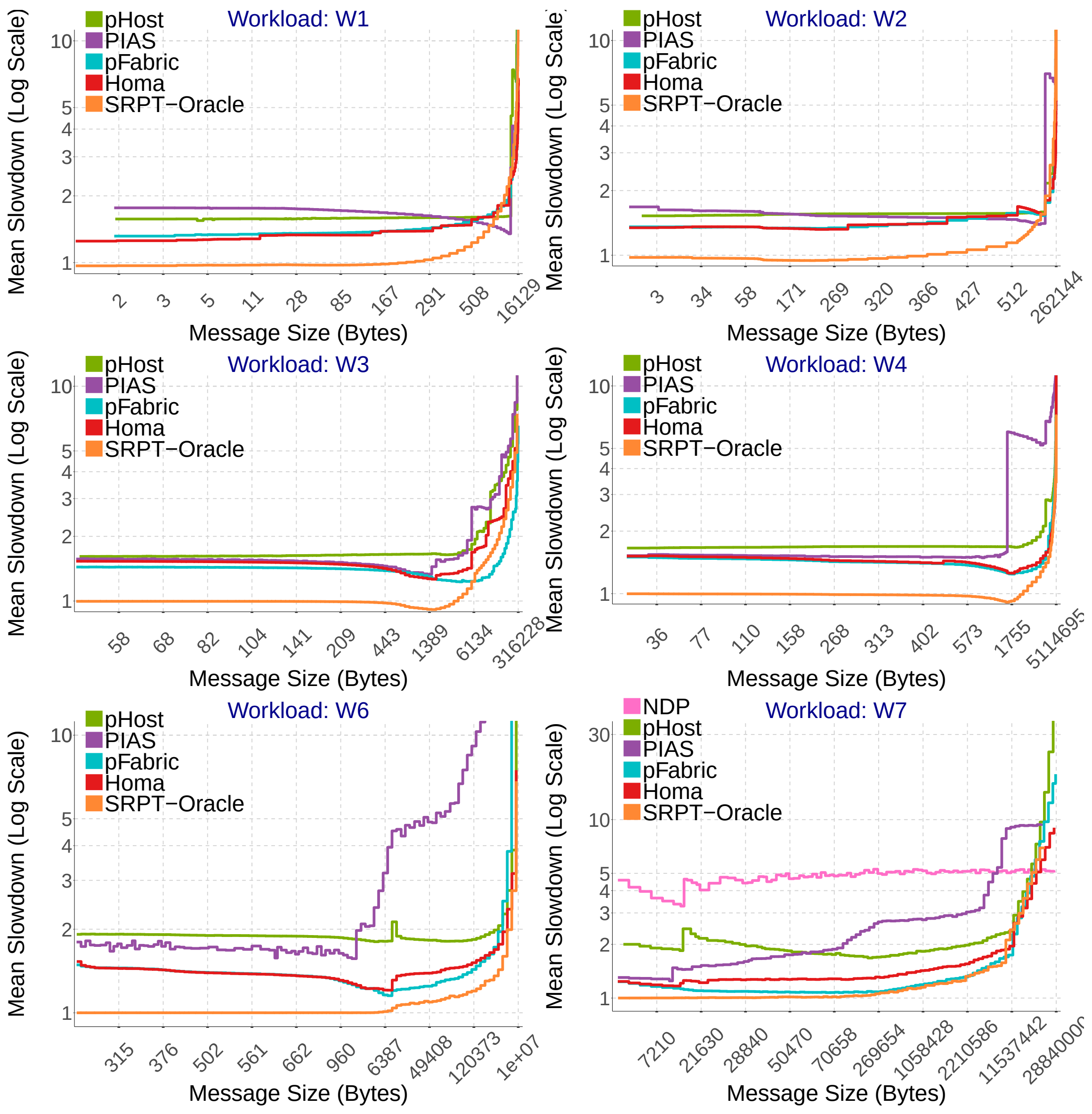


Figure 6.12: Mean slowdown as a function of message size, at 80% network load, for different protocols and workloads. Distance on the x-axis is linear in total number of messages (each tick corresponds to 10% of all messages). Graphs were measured at 80% network load, except for NDP and pHost. Neither NDP nor pHost can support 80% network load for these workloads, so we used the highest load that each protocol could support (70% for NDP, 58–73% for pHost, depending on workload). The minimum one-way time for a small message (slowdown is 1.0) is 2.3 µs. NDP was measured only for W7 because its simulator cannot handle partial packets.

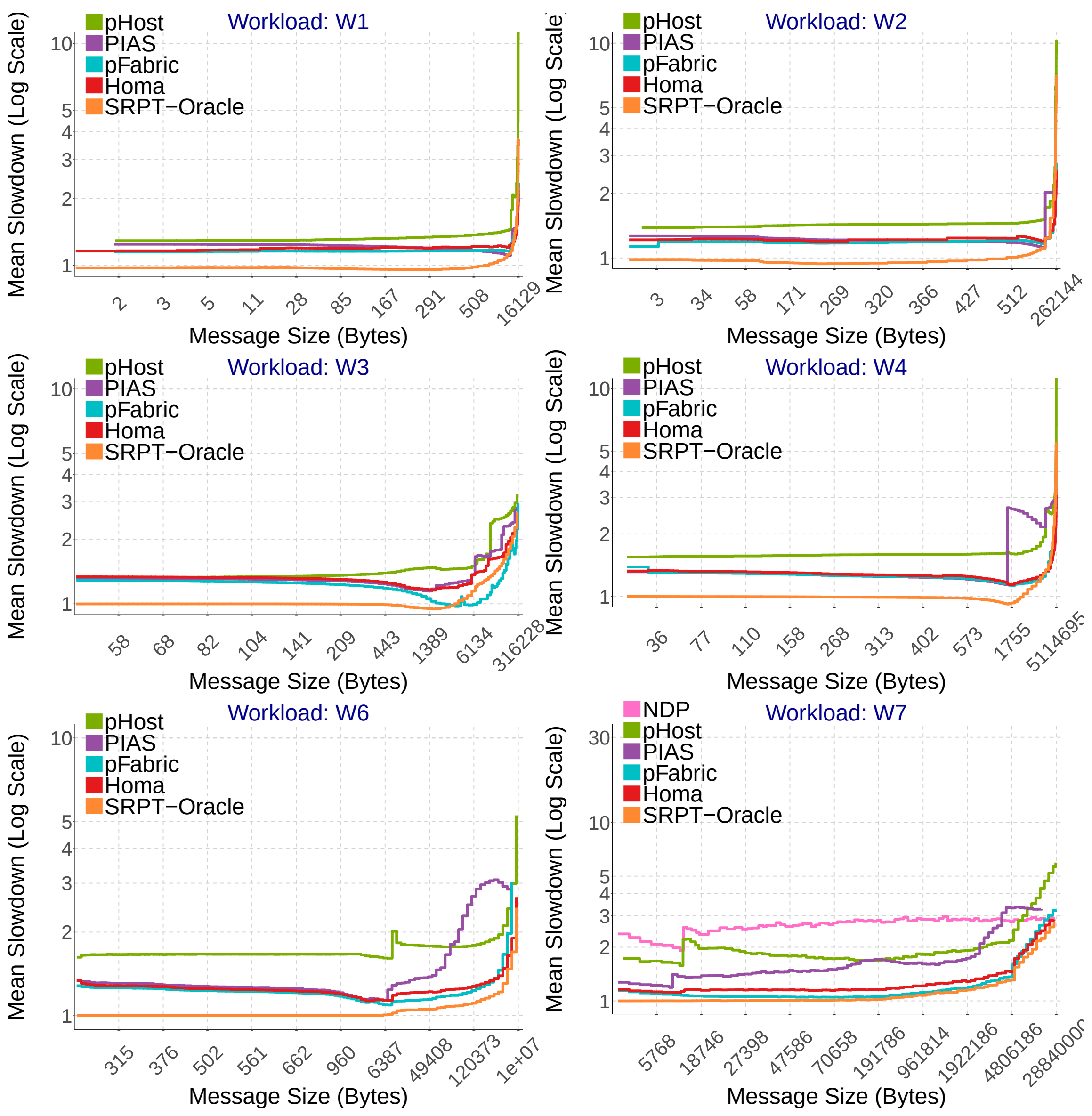


Figure 6.13: Mean slowdown as a function of message size, at 50% network load, for different protocols and workloads. Distance on the x-axis is linear in total number of messages (each tick corresponds to 10% of all messages). Graphs were measured at 50% network load. The minimum one-way time for a small message (slowdown is 1.0) is 2.3 µs. NDP was measured only for W7 because its simulator cannot handle partial packets.

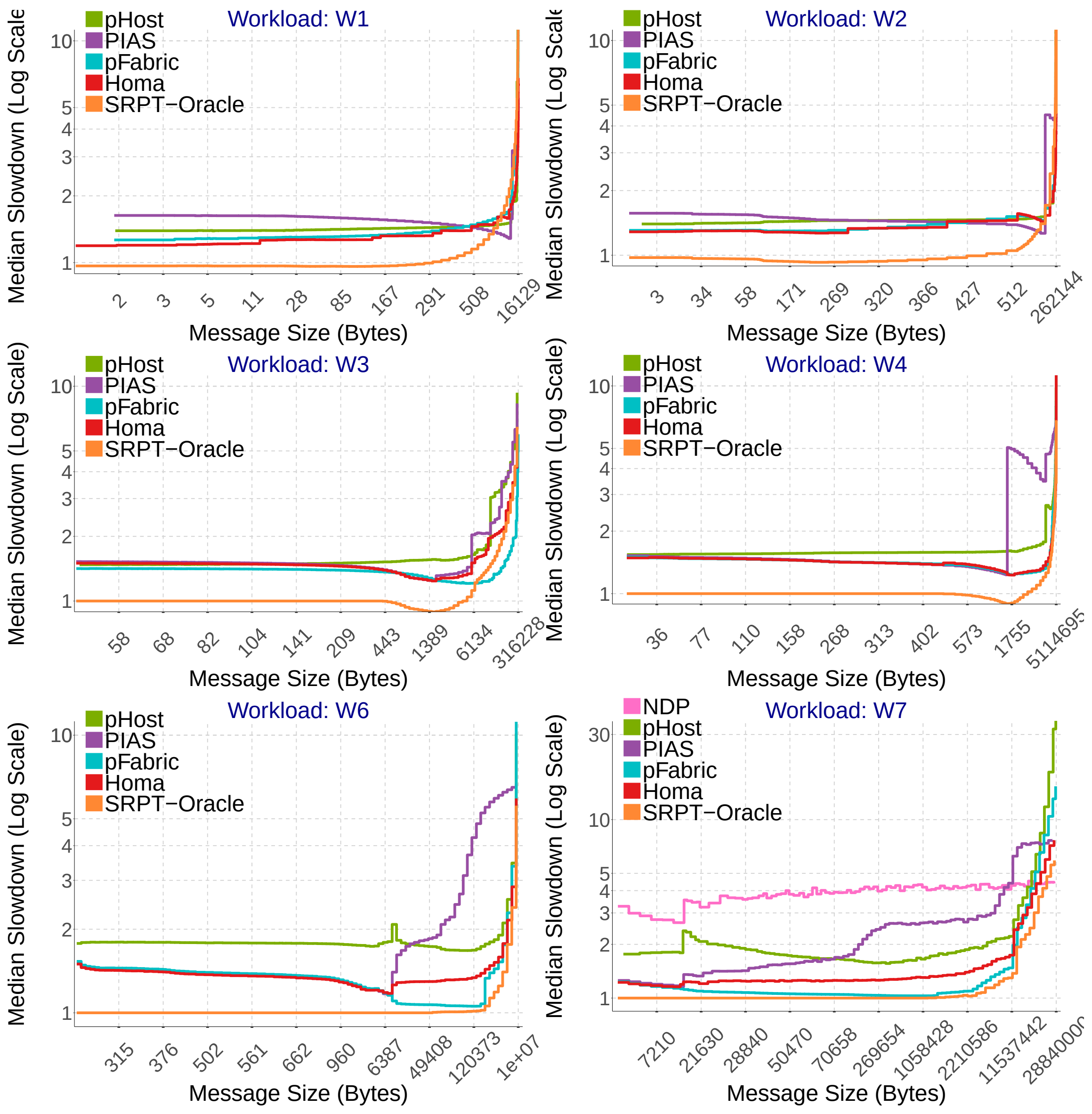


Figure 6.14: Median slowdown as a function of message size, at 80% network load, for different protocols and sssssssorkloads. Distance on the x-axis is linear in total number of messages (each tick corresponds to 10% of all messages). Graphs were measured at 80% network load, except for NDP and pHost. Neither NDP nor pHost can support 80% network load for these workloads, so we used the highest load that each protocol could support (70% for NDP, 58–73% for pHost, depending on workload). The minimum one-way time for a small message (slowdown is 1.0) is 2.3 µs. NDP was measured only for W7 because its simulator cannot handle partial packets.

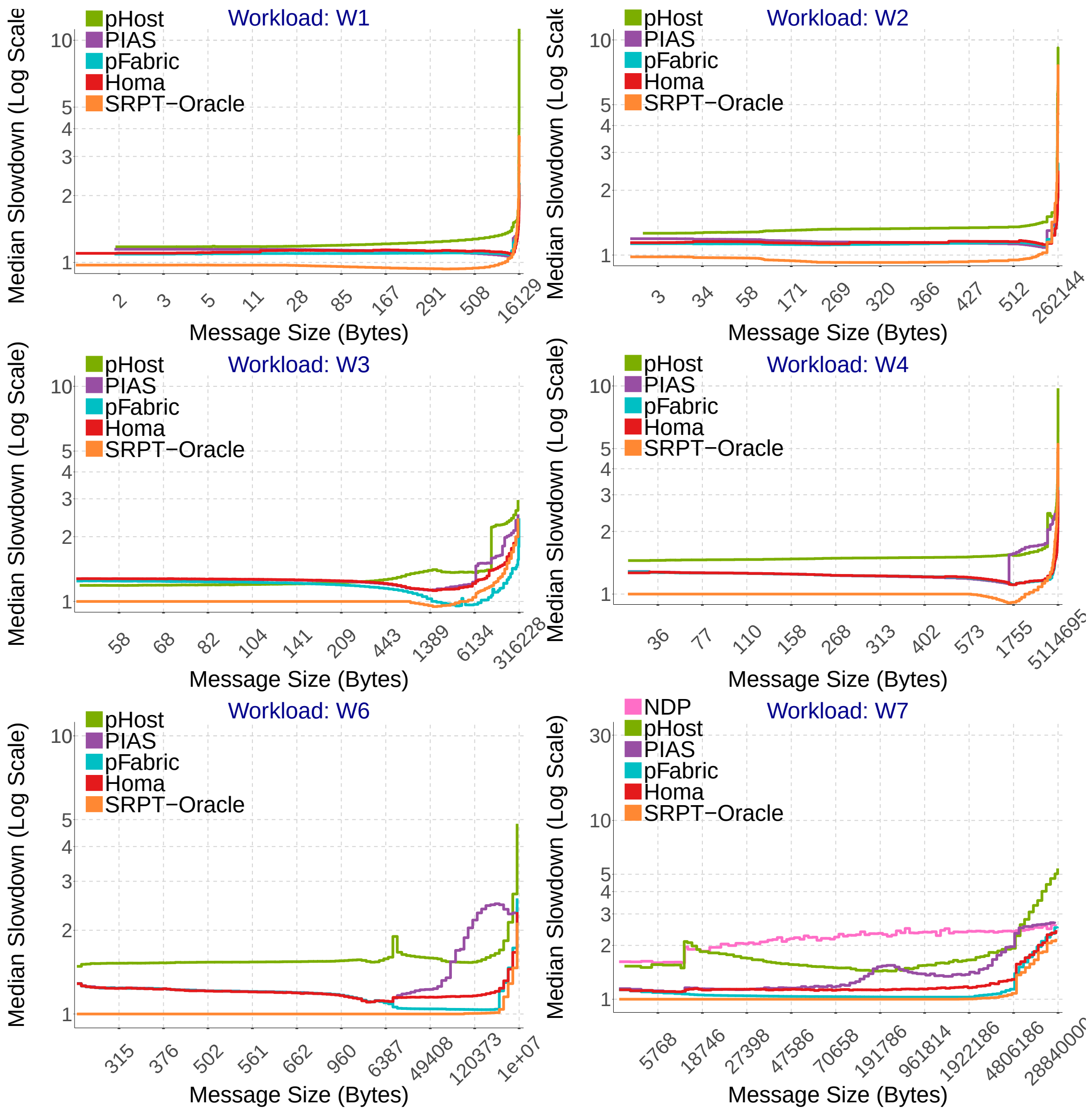


Figure 6.15: Median slowdown as a function of message size, at 50% network load, for different protocols and workloads. Distance on the x-axis is linear in total number of messages (each tick corresponds to 10% of all messages). The minimum one-way time for a small message (slowdown is 1.0) is 2.3 µs. NDP was measured only for W5 because its simulator cannot handle partial packets.

Chapter 7

System Implementation And Evaluation

We implemented Homa as a new transport in the RAMCloud main-memory storage system [27] and measured Homa's performance in a real test bed setup. The purpose of this experiment is to validate our simulation results of Chapter 6 and confirm that Homa can indeed achieve its low latency and high throughput goal in real application.

RAMCloud is a great platform for Homa's implementation because it's a low latency system. RAMCloud supports a variety of transports that use different networking technologies, and it has a highly tuned software stack: the total software overhead to send or receive an RPC is 1–2 µs in most transports. The Homa transport is based on DPDK [10], which allows it to bypass the kernel and communicate directly with the NIC; Homa detects incoming packets with polling rather than interrupts. The Homa implementation contains a total of approximately 4000 lines of C++ code, of which about half are comments.

7.1 Implementing Transports For RPCs

The RAMCloud implementation of Homa includes all of the features described in Chapters 3 and 5. Specifically, to achieve SRPT the implementation includes receiver-driven flow control, packet priority assignment scheme, and traffic pacers at senders; to achieve high bandwidth utilization, it has the overcommitment mechanism. These features were the key ideas in Homa's design and were pivotal to Homa's transport mechanism and its performance. So, we allocated the majority of the previous chapters to designing and analysing them in simulations and we explained how to implement them in practice. The only thing that is not implemented in the RAMCloud implementation of Homa is that it does not yet measure incoming message lengths on the fly (the priorities were

DATA	Sent from sender to receiver. Contains a range of bytes within a message, defined by an offset and a length. Also indicates total message length.
GRANT	Sent from receiver to sender. Indicates that the sender may now transmit all bytes in the message up to a given offset, and specifies the priority level to use.
RESEND	Sent from receiver to sender. Indicates that sender should re-transmit a given range of bytes within a message.
BUSY	Sent from sender to receiver. Indicates that a response to RESEND will be delayed (the sender is busy transmitting higher priority messages, or an RPC operation is still being executed); used to prevent timeouts.

Figure 7.1: The packet types used by Homa. All packet types except DATA are sent at highest priority; the priorities for DATA packets are specified by the receiver as discussed in Chapter 3.

precomputed based on knowledge of the benchmark workload).

In addition to the key ideas from the previous chapters, the RAMCloud implementation of Homa contains several unusual features: it is message-oriented, rather than stream-oriented; it is connectionless; it uses no explicit acknowledgments; it implements timeouts and packet retransmissions (simulations of previous chapter didn't implement this); it has a mechanism to handle incast scenarios; and it implements at-least-once semantics, rather than the more traditional at-most-once semantics. In this section we discuss these other aspects of the protocol that most of them are less essential for latency performance but result in a complete and practical substrate for datacenter RPC.

Homa's RAMCloud implementation uses four packet types, which are summarized in Figure 7.1. The DATA type is used for unscheduled and scheduled packets, the GRANT type as the name suggests is for grant packets, RESEND is used when a packet retransmission is requested by the receiver, and BUSY is a control packet that is to avoid retransmission time outs at the receivers. Later in this section, we will explain the last two types in more detail.

7.1.1 Homa: Transport For RPCs, Not Connections

What is RPC? An RPC consists of a request message from a client to a server and its corresponding response message. RPCs have a ubiquitous presence in datacenters; they are the primary type of communication primitives used in datacenters. The majority of the distributed datacenter applications that we know of use RPCs to communicate with remote servers. For example in a distributed key-value store, when a client application performs a read from a remote storage server, it uses an RPC to get the data. In such RPCs, the request includes the key and is sent from the application to the server and the server sends back the value corresponding to that key in the response of the RPC. Each RPC is identified by a globally unique RPCid generated by the client. The RPCid is included in all packets associated with the RPC. An RPC is considered finished when the response is delivered to the application in its entirety. The client has to keep the state associated with an RPCid for as long as the RPC is not finished. A client may have any number of outstanding unfinished RPCs at a time, to any number of servers.

One of the unique features of Homa is that it doesn't support connections; instead it is a transport for RPCs. No setup phase or connection is required before a client initiates an RPC to a server, and neither the client nor the server retains any state about an RPC once the client has received the result. Homa's connectionless approach means that the state kept on a server is determined by the number of active RPCs, not the total number of clients and this significantly reduces the state that needs to be kept. In datacenter applications, servers can have large numbers of clients; for example, servers in Google datacenters commonly have several hundred thousand open connections, at least one per clients [13]. With connection oriented transports, this means the server needs to maintain lots of state for each client. In contrast, Homa's approach to keep state only for active RPCs significantly reduces the amount of state that needs to be kept.

Homa's RPC approach, contrast to traditional streaming approaches, allows out of order delivery of messages; out of order delivery of messages between a sender-receiver pair is critical for low tail latency. When a short message is presented to a sender for transmission to a receiver, after a long message between the same sender-receiver pair, we don't want the longer message to block the quick delivery of the shorter one. Homa's RPC approach naturally realizes the out of order delivery behavior because concurrent RPCs to the same server may complete in any order. This is contrast to the streaming approach used by TCP which serializes all messages between the same sender-receiver pair based on the order they arrive at the sender. This serialization results in head-of-line-blocking, where a short message is queued behind a long message for the same destination. Later in this chapter will show that the streaming approach increases tail latency by 100x for short messages. We need to also mention that some recent stream-based proposals, such as DCTCP, pFabric, and PIAS, assume dozens of connections between each sender-receiver pair, so that each message has a dedicated connection to avoid serialization of messages on the same stream. However, this approach results in an explosion of connection state. As we discussed previously, even a single connection for each application-server pair is problematic for large-scale applications ([24], [12]), so it is probably not realistic to use multiple connections. So these protocols are most likely not viable as described.

Another benefit of Homa's RPC approach is that it eliminates the need for acknowledgment packets. Homa doesn't use explicit acknowledgment packets as in TCP or many other protocols because each RPC request inevitably requires a response. Homa uses the response as an acknowledgment for the request. This reduces the number of packets transmitted over the network (in the simplest case, there is only a single request packet and a single response packet). One-way messages can be simulated by having the server application return an empty response immediately upon receipt of the request. This design decision of not using acknowledgment packets has implications on how to detect and retransmit lost packets which is the topic of the next section.

Although we designed Homa for newer datacenter applications where RPC is a natural fit, we believe that traditional applications could be supported by implementing a socket-like byte stream interface above Homa. We leave this for future work.

7.1.2 Retransmission of Lost Packets

We expect lost packets to be rare in Homa. There are two reasons for packet loss: corruption in the network, and buffer overflow. Corruption is extremely rare in modern datacenter networks, and Homa reduces buffer usage enough to make buffer overflows extremely uncommon as well (refer to Section 6.4.2). Since packets are almost never lost, Homa optimizes lost-packet handling for efficiency in the common case where packets are not lost, and for simplicity (as opposed to efficiency) when packets are lost.

In TCP, senders are responsible for detecting lost packets. This approach requires acknowledgment packets, which add overhead to the protocol (the simplest RPC requires two data packets and two acknowledgments). In Homa, receivers are responsible for detecting the lost packets; as a result, Homa does not use any explicit acknowledgments. This eliminates half of the packets for simple RPCs. Receivers use a simple timeout-based mechanism to detect lost packets. If a long time period (a few milliseconds) elapses without additional packets arriving for a message, the receiver sends a RESEND packet that identifies the first range of missing bytes; the sender will then retransmit those bytes.

If all of the initial packets of an RPC request are lost, the server will not know about the message, so it won't issue RESENDs. However, the client will timeout on the response message, and it will send a RESEND for the response (it does this even if the request has not been fully transmitted). When the server receives a RESEND for a response with an unknown RPCid, it assumes that the request message must have been lost and it sends a RESEND for the first RTTbytes of the request.

If a client receives no response to a RESEND (because of server or network failures), it retries the RESEND several times and eventually aborts the RPC, returning an error to higher level software.

7.1.3 Controlling Incast

Incast is a type of traffic pattern that can cause buffer overflow in a receiver's TOR switch and cause massive reduction in bandwidth utilization at the receiver downlink. In incast, multiple senders transmit simultaneously to the same receiver and each sends enough packets such that they consume all of the buffer space in the TOR near the receiver. Under this condition, lost packets won't be rare anymore as incast causes massive packet drops in the TOR for all of the senders' messages. This massively cripples the progress of the transport as it causes a very large number of retransmission timeouts and retransmission requests for all messages. The net effect is a significant reduction in bandwidth utilization because 1) the timeouts are long and no packet will be retransmitted until the timeout occurs and 2) a lot of bandwidth get wasted as a result of many packets that drop. Hence none of the messages can make progress to completion in a timely manner.

Incast is usually self inflicted; it occurs when a node issues many concurrent RPCs to many servers, all of which return their results at the same time. The simultaneous arrival of these responses causes incast at the TOR switch near the node that issued the RPCs.

Homa solves the incast problem by taking advantage of the fact that incast is usually self-inflicted: Homa detects impending incasts by counting each node's outstanding RPCs. Once this number exceeds a threshold, new RPCs are marked with a special flag that causes the server to use a lower limit for unscheduled bytes in the response message (a few hundred bytes). Small responses will still get through quickly, but larger responses will be scheduled by the receiver; the overcommitment mechanism will limit buffer usage. With this approach, a 1000-fold incast will consume at most a few hundred thousand bytes of buffer space in the TOR.

Incast can also occur in ways that are not predictable; for example, several machines might simultaneously decide to issue requests to a single server. However, it is unlikely that many such requests will synchronize tightly enough to cause incast problems. If this should occur, Homa's efficient use of buffer space still allows it to support hundreds of simultaneous arrivals without packet loss. We will show this later in our evaluation of incast in this chapter.

Incast is largely a consequence of the high latency in current datacenters. If each request results in a disk I/O that takes 10 ms, a client can issue 1000 or more requests before the first response arrives, resulting in massive incast. In future low-latency environments, incast will be less of an issue because requests will complete before very many have been issued. For example, in the RAMCloud main-memory storage system [27], the end-to-end round-trip time for a read request is about 5μs. In a multiread request where a client issues multiple concurrent read requests to different servers, it takes the client 1–2μs to issue each request; by the time it has issued 3–4 RPCs, responses from the first requests have begun to arrive. Thus there are rarely more than a few outstanding requests.

7.1.4 At-least-once Semantics

RPC protocols have traditionally implemented at most once semantics, where each RPC is executed exactly once in the normal case; in the event of an error, an RPC may be executed either once or not at all. In contrast, Homa allows RPCs to be executed more than once: in the normal case, an RPC is executed one or more times; after an error, it could have been executed any number of times (including zero). There are two situations where Homa re-executes RPCs. First, Homa doesn't keep connection state, so if a duplicate request packet arrives after the server has already processed the original request and discarded its state, Homa will re-execute the operation. Second, servers get no acknowledgment that a response was received, so there is no obvious time at which it is safe to discard the response. Since lost packets are rare, servers take the simplest approach and discard state for an RPC as soon as they have transmitted the last response packet. If a response packet is lost, the server may receive the RESEND after it has deleted the RPC state. In this case, it will behave as if it never received the request and issue a RESEND for the request; this will result in re-execution of the RPC.

Homa allows re-executions because it simplifies the implementation and allows servers to discard all state for inactive clients (at-most-once semantics requires servers to retain enough state for each client to detect duplicate requests). Moreover, duplicate suppression at the transport level is insufficient for most datacenter applications. For example, consider a replicated storage system: if a particular replica crashes while executing a client's request, the client will retry that request with a different replica. However, it is possible that the original replica completed the operation before it crashed. As a result, the crash recovery mechanism may result in re-execution of a request, even if the transport implements at-most-once semantics. In general, duplicates must be filtered at a level above the transport layer.

Homa assumes that higher level software will either tolerate redundant executions of RPCs or filter them out. The filtering can be done either with application-specific mechanisms, or with general-purpose mechanisms such as RIFL [20]. For example, a TCP-like streaming mechanism can be implemented as a very thin layer on top of Homa that discards duplicate data and preserves order.

	CloudLab	Infiniband
CPU	Xeon D1548 (8 cores @ 2.0 GHz)	Xeon X3470 (4 cores @ 2.93 GHz)
NICs	Mellanox ConnectX-3 (10 Gbps Ethernet)	Mellanox ConnectX-2 (24 Gbps)
Switches	HP Moonshot-45XGc (10 Gbps Ethernet)	Mellanox MSX6036 (4X FDR) and Infiniscale IV (4X QDR)

Figure 7.2: Hardware configurations used in experiments with Homa's implementation. The Infiniband cluster was used only for measuring Infiniband performance; CloudLab was used for all other measurements.

7.2 Implementation Measurements

Our goal with the RAMCloud implementation of Homa is to see if a practical implementation can provide the same benefits we saw in the simulations. Similar to the Homa simulations in chapter 6, we'd like to answer the following questions in this section:

1. Does Homa provide low latency for short messages even at high network load and in the presence of long messages?
2. How efficiently does Homa use network bandwidth?
3. How does Homa compare to existing low latency approaches used in practice?
4. How important are Homa's novel features to its performance?

We used the CloudLab cluster described in Figure 7.2 to measure the performance of the Homa implementation in RAMCloud. The cluster contained 16 nodes connected to a single switch using 10 Gbps Ethernet; 8 nodes were used as clients and 8 as servers. Each client generated a series of echo RPCs; each RPC sent a block of a given size to a server, and the server returned the block back to the client. Clients chose RPC sizes pseudo-randomly to match one of the workloads from Figure 4.2(a), with Poisson arrivals configured to generate a particular network load. The server for each RPC was chosen at random.

The Homa implementation experiments are different from the simulations of Chapter 6 in two aspects: first, the simulations only focused on the network latencies and didn't consider latency overheads and queuing delays inflicted by the end-host software stack. Conversely, the implementation measurements by default include the variable delays incurred by the software and CPUs in packet and protocol processing, PCI-e communication and memory copies. Secondly, in the simulations, latencies were measured in terms of one way delays from when a message is presented to the sender's transport until it's fully received by the receiver. This is in contrast to the implementation experiments where the latencies are measured in terms of round-trip times from when the request of an RPC is presented to the client's transport until the response is fully received by the client.

7.2.1 Homa Performance Analysis

Figures 7.3, 7.4, and 7.5 show the performance of Homa and several variations of Homa for workloads W2-W7 at 80% network load. Our primary metric for evaluating Homa, shown in Figure 7.3, is 99th percentile tail slowdown, where a slowdown of 1 is ideal and lower numbers are better. The figures show slowdown spectrum plots where the x-axis is linear in the total number of messages, with ticks corresponding to 10% of all messages in that workload. This results in a different x-axis scale for each workload, which makes it easier to see results for the message sizes that are most common. This benefits of this type of plot were discussed in details in Section 4.3.1.

In the figures, W1 is not shown because RAMCloud's software overheads are too high to handle the large numbers of small messages generated by this workloads at 80% network utilization. Furthermore, W2 is shown at 41% network load, which is the maximum load we could utilize with this workload; beyond 41% network load, the experiment becomes unstable and the input messages queues grow out of bound.

Homa provides a 99th percentile tail slowdown in the range of 1.8–3.1 across a broad range of RPC sizes and workloads. For example, a 100-byte echo RPC takes 4.7 µs in an unloaded network. At 80% network load, the 99th-percentile latency was about 14 µs (corresponding to slowdown of 3) in all 6 workloads. Compared to the simulations results, tail slowdown is a bit larger with the implementation results; in the simulations, the minimum one-way transmission time for a 100-byte message is 2.1µs and 99th-percentile latency is 4.8µs (corresponding to slowdown of 2.3). The extra slowdown in the implementation compared to the simulations is a result of queuing and packet processing delays in the software.

To quantify the benefits provided by the priority and overcommitment mechanisms in Homa, we also measured RAMCloud's Basic transport. Basic is similar to Homa in that it is receiver-driven, with grants and unscheduled packets. However, Basic does not use priorities and it has no limit on overcommitment: receivers grant independently and simultaneously to all incoming messages. Figure 7.3 shows that tail latency is 5–15x higher in Basic than in Homa for most of the workloads. By analyzing detailed packet traces we determined that Basic's high latency is caused by queuing

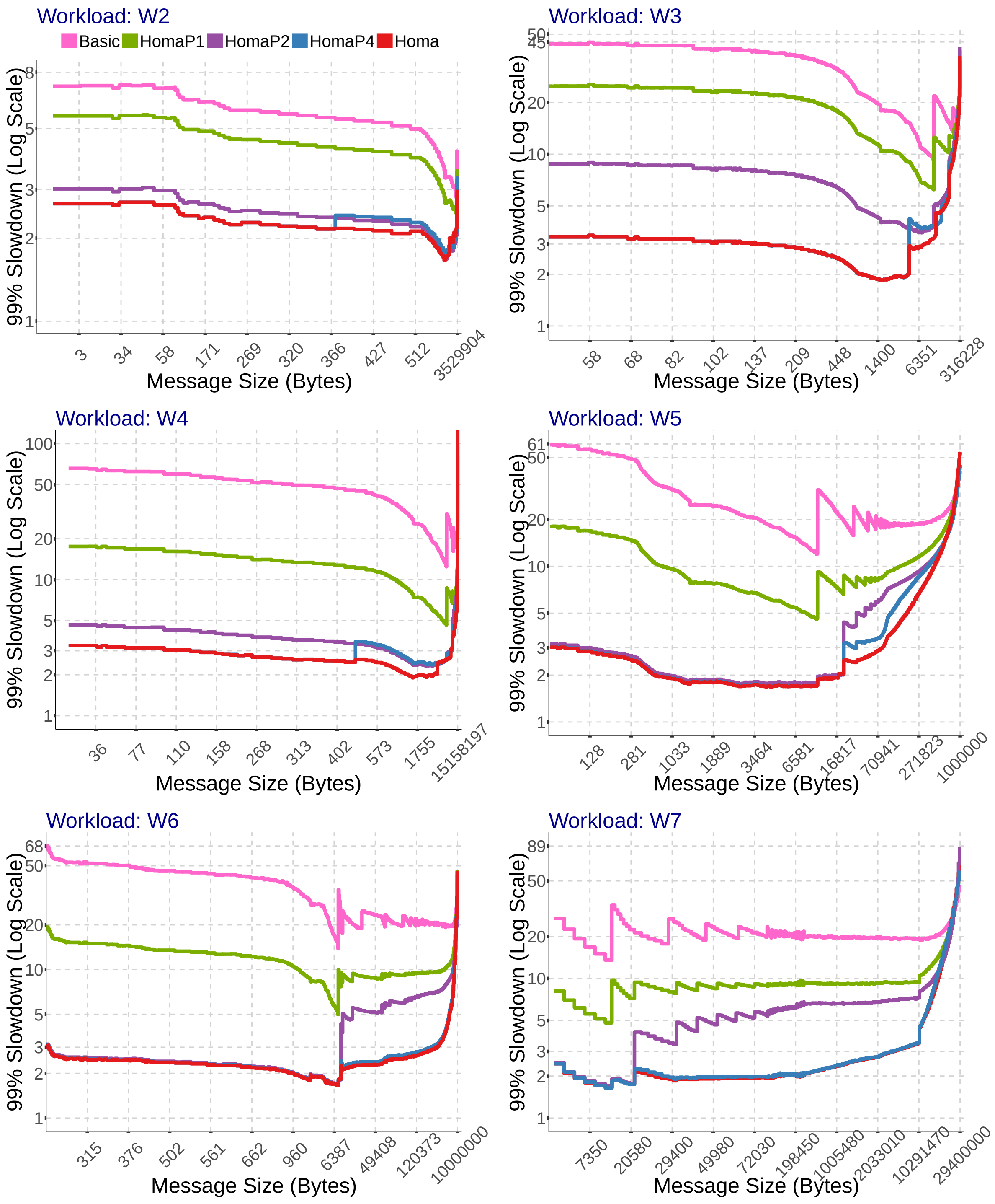


Figure 7.3: Tail latency of Homa and its variations for workloads W2–W7 at 80% network load, except W2 that is ran at 41% network load. X-axes are linear in total number of messages (each tick is 10% of all messages). “HomaPx” measures Homa restricted to use only x priorities. “Basic” measures the preexisting Basic transport in RAMCloud, which corresponds roughly to HomaP1 with no limit on overcommitment. Best-case RPC times (slowdown of 1.0) for 100 byte RPCs are 4.7 µs for Homa and Basic.

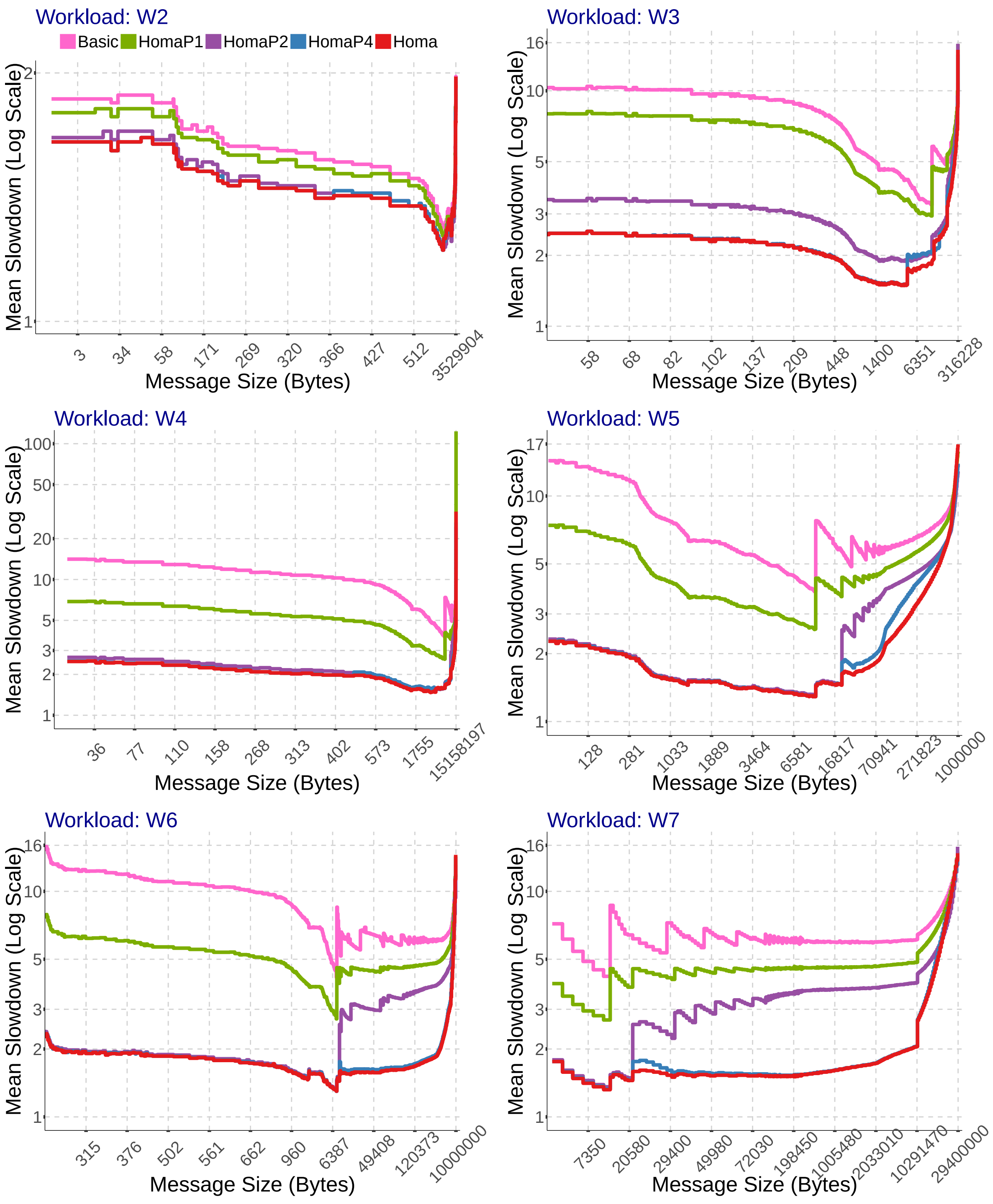


Figure 7.4: same as figure 7.3 except the y-axis is mean slowdown instead of 99th percentile slowdown.

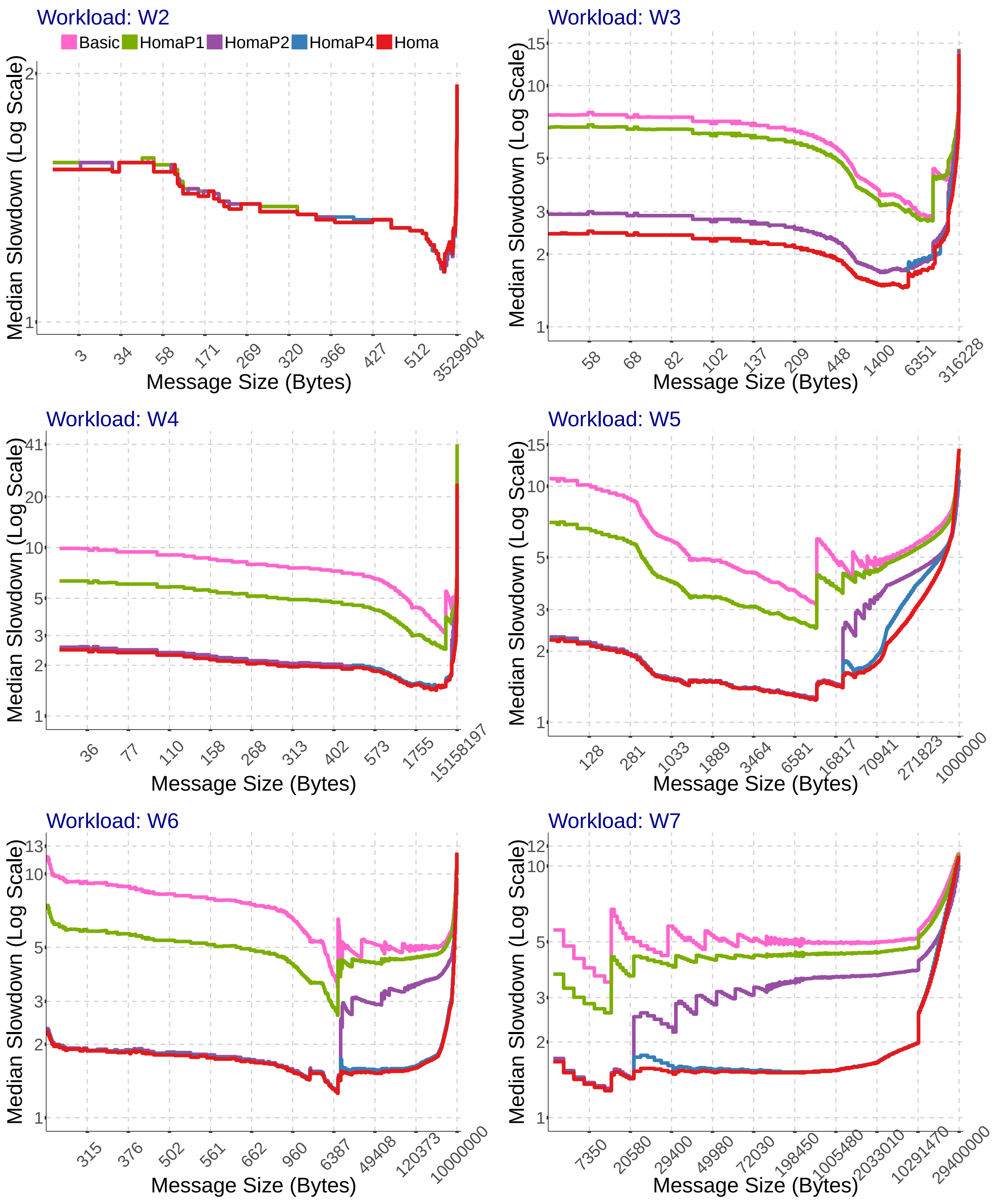


Figure 7.5: Same as Figure 7.3 except the y-axis is median slowdown instead of 99th percentile slowdown.

delays at the receiver's downlink; Homa's use of priorities eliminates almost all of these delays.

Although Homa prioritizes small messages, it also outperforms Basic for large ones. This is because Homa's SRPT policy tends to produce run-to-completion behavior: it finishes the highest priority message before giving service to any other messages. In contrast, Basic, like TCP, tends to produce round-robin behavior; when there are competing large messages, they all complete slowly.

For the very largest messages, Homa produces 99th-percentile slowdowns between 30–100x depending on the workload. This is very similar to what we observed in the simulations of Chapter 6, where the worst tail slowdown were between 11x to 33x. This is because of the SRPT policy; in SRPT the largest messages are at disadvantage for scheduling priority. As we discussed in previous chapter, we speculate that the performance of these outliers could be improved by dedicating a small fraction of downlink bandwidth to the oldest message; we leave a full analysis of this alternative to future work.

To answer the question "How many priority levels does Homa need?" we modified the Homa transport to reduce the number of priority levels by collapsing adjacent priorities while maintaining the same degree of overcommitment. Figures 7.3 and 7.5 show the results. 99th-percentile tail latency is almost as good with 4 priority levels as with 8, but tail latency increases noticeably when there are only 2 priority levels. Even when considering median slowdown (Figure 7.5), performance is considerably better with two priorities than just one. Homa with only one priority level is still significantly better than Basic; this is because Homa's limit on overcommitment results in less buffering than Basic, and this reduces preemption lag.

7.2.2 Homa vs. Infiniband

We also compared Homa against RAMCloud's InfRC transport, which uses kernel bypass with Infiniband reliable connected queue pairs.

Figures 7.6 shows the comparison between Homa and Infiniband only for workloads W4, W6, and W7. Other workloads had too many short messages; high software overheads didn't allow us to run Infiniband with them.

The Infiniband measurements were taken on a different cluster with faster CPUs, and the Infiniband network has 24 Gpbs application level bandwidth, vs. 10 Gbps for Homa and Basic. The software overheads for InfRC were too high to run at 80% load on the Infiniband network; with 2.4x more bandwidth than the 10Gbps Ethernet used for Homa, it would take 2.4x more messages/sec to achieve 80% load. Therefore, we used the same absolute load as for the Homa measurements, which resulted in $\underline{\text{only 33\% network load for Infiniband}}$ (i.e. $\frac{80\%}{2.4} = 33\%$). This give a massive benefit to

Infiniband in our comparisons because it's running at a much lower network load.

The Infiniband measurements show the advantage of Homa's message-oriented protocol over streaming protocols. We first measured InfRC in its normal mode, which uses a single connection for each client-server pair. This resulted in tail latencies about 1000x higher than Homa for small messages. Detailed traces showed that the long delays were caused by head-of-line blocking at the sender, where a small message got stuck behind a very large message to the same destination. Any streaming protocol, such as TCP, will suffer similar problems because of head-of-line blocking; this validates the advantage of Homa's RPC approach to the streaming approaches as we discussed in Section 7.1.1. Infiniband is widely believed to be a low latency mechanism, but in fact our experiments show that it doesn't provide good tail latency at high network load.

We modified the Infiniband benchmark to use multiple connections per client-server pair ("InfRC-MC" in the figures). This eliminated the head-of-line blocking and improved tail latency by 100x, to about the same level as Basic. As discussed in Section 7.1.1 in this chapter, this approach is probably not practical in large-scale applications because it causes an explosion of connection state. InfRC-MC still doesn't approach Homa's performance, because it doesn't use priorities.

As a result of lower network load for Infiniband experiments (33% vs Homa's 80%), the figure overstates the performance of Infiniband relative to Homa. In particular, Infiniband appears to perform better than Homa for large message sizes. This is an artifact of measuring Infiniband at 33% network load and Homa at 80%; at equal load factors, we expect Homa to provide significantly lower latency than Infiniband at all message sizes.

7.3 Homa vs. TCP

The "TCP-MC" curves in Figure 7.6 show the performance of RAMCloud's TCP transport, which uses the Linux kernel implementation of TCP. Only workloads W6 and W7 are shown (system overheads were too high to run other workloads at 80% load), and only with multiple connections per client-server pair (with a single connection, tail slowdown was off the scale of the graphs). Even in multi-connection mode, TCP's tail latencies are 10–100x higher than for Homa and TCP is also a lot slower than Infiniband. We also created a new RAMCloud transport using mTCP [19], a user-level implementation of TCP that uses DPDK for kernel bypass. However, we were unable to achieve latencies for mTCP less than 1 ms; the mTCP developers confirmed that this behavior is expected (mTCP batches heavily, which improves throughput at the expense of latency). We did not graph mTCP results.

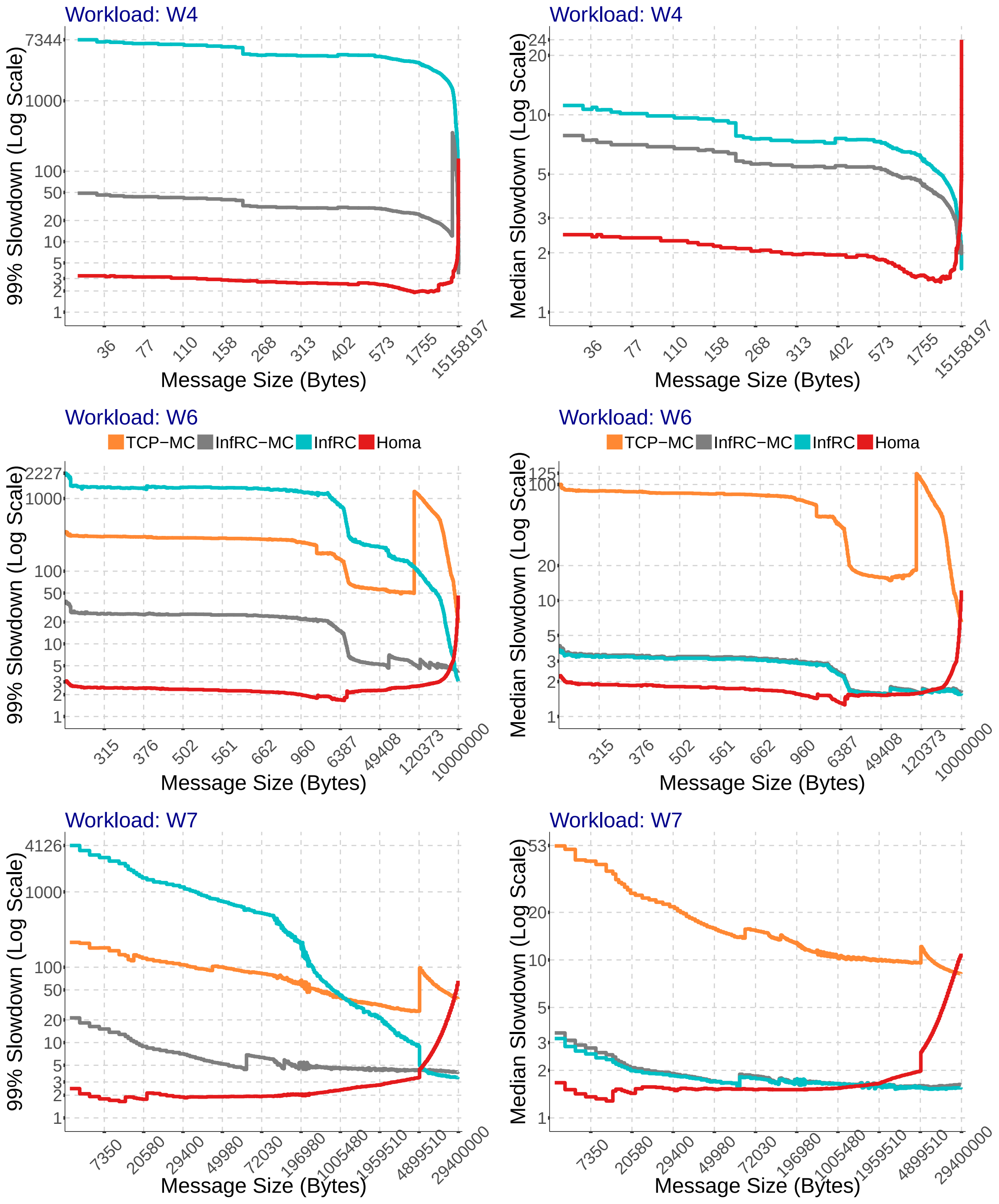


Figure 7.6: Tail and median latency of Homa, Infiniband and TCP for workloads W4, W6, and W7 at 80% network load, ”InfRC” measures RAMCloud’s Infrc transport, which uses Infiniband reliable connected queue pairs. “InfRC-MC” uses Infiniband with multiple connections per client-server pair. “TCP-MC” uses kernel TCP with multiple connections per client-server pair. Homa was measured on the CloudLab cluster. InfRC was measured on the Infiniband cluster using the same absolute workload, so its network utilization was only about 33%. Best-case RPC times (slowdown of 1.0) for 100 byte RPCs are 4.7 µs for Homa. 3.9 µs for InfRC, 15.5 µs for TCP.

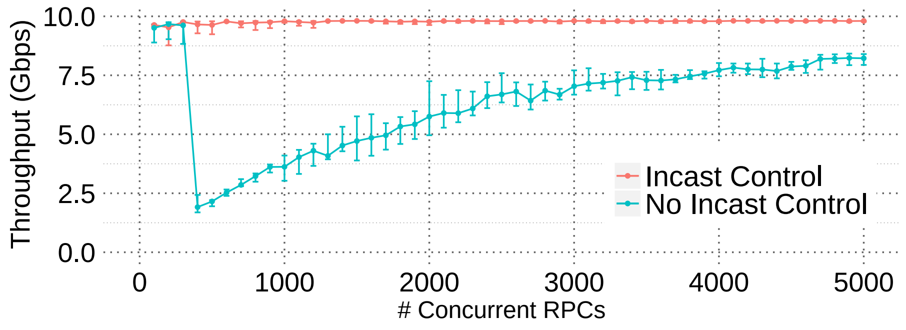


Figure 7.7: Homa's performance under incast. The figure shows overall throughput when a single Homa client receives responses for RPCs issued concurrently to 15 servers. Each response was 10 KB. Each data point shows min, mean, and max values over 10 runs. When the incast control mechanism is disabled, Homa can support up to 300 concurrent responses without throughput degradation. With incast control enabled, Homa is able to support an incast of several thousand concurrent transmissions.

7.4 Homa vs. Other Implementations

It is difficult to compare Homa with other published implementations because most prior systems do not break out small message performance and some measurements were taken with slower networks. Nonetheless, Homa's absolute performance (14 µs round-trip for small messages at 80% network load and 99th percentile tail latency) is nearly two orders of magnitude faster than the best available comparison systems. For example, HULL [4] reported 782 µs one-way latency for 1 Kbyte messages at 99th percentile and 60% network load, and PIAS [7] reported 2 ms one-way latency for messages shorter than 100 Kbytes at 99th percentile and 80% network load; both of these systems used 1 Gbps networks. NDP [16] reported more than 600 µs one-way latency for 100 Kbyte messages at 99th percentile in a loaded 10 Gbps network, of which more than 400 µs was queueing delay.

7.4.1 Homa Performance Under Incast

To measure the effectiveness of Homa's incast control mechanism, we ran an experiment where a single client initiated a large number of RPCs in parallel to a collection of servers; this causes self inflicted incast scenario for the client. In Homa, incast can only happen as a result of a flood of uninvited unscheduled packets. Therefore, in this experiment each RPC had a tiny request and a response of approximately RTTbytes (10 KB) that fully fits in unscheduled packets (larger responses would produce similar incast). Figure 7.7 shows the results. With the incast control mechanism enabled, Homa successfully handled several thousand simultaneous RPCs without degradation.

We also measured performance with incast control disabled; this shows the performance that can be expected when incast occurs for unpredictable reasons. Beyond 300 concurrent RPCs, the utilized bandwidth of the client's downlink drops which means the buffer space in the TOR is full and packets are dropping because of incast (throughput drops because of long transmission timeouts and wasted bandwidth from packet drops). This demonstrates that under unpredictable incast condition, Homa supports about 300 concurrent RPCs before performance degrades. Homa is less sensitive to incast than protocols such as TCP because its packet scheduling mechanism limits buffer buildup to at most RTTbytes per incoming message. In contrast, a single TCP connection can consume all of the buffer space available in a switch. Note that in the curve for disabled incast control, throughput rises again when number of concurrent RPCs increases beyond 400. This is because as the number of RPCs increases, the number of timeouts that happens in a fixed period time increases and more number of RPCs make progress toward completion.

7.4.2 Homa Under Lower Loads

For curious readers, we also included the slowdown of Homa and its variations under moderate and lower network loads. Figures 7.8, 7.9, and 7.10 show tail, mean and median slowdown at moderate 50% network load and Figures 7.11, 7.12, and 7.13 show these metrics at low 35% load. Similar trends as in 80% network load plots can be observed in these plots. Homa outperforms other schemes in tail slowdown; regardless of the network load, both priorities and controlled overcommitment are critical for good tail and even mean latency of short messages. However, at lower loads the median latency for short messages is less sensitive to the number of priorities or controlled overcommitment. That is because at lower loads, network queues are empty most of the time and head-of-line blocking happens less frequently.

7.5 Chapter Summary

In this chapter we discussed an implementation of Homa in the RAMCloud storage system. We introduced several new features of the RAMCloud's implementation of Homa, in addition to the key ideas presented in previous chapters. These new features includes Homa is message-oriented, rather than stream-oriented; it is connectionless; it uses no explicit acknowledgments; it implements timeouts and packet retransmissions (simulations of previous chapter didn't implement this); it has a mechanism to handle incast scenarios; and it implements at-least-once semantics.

We showed that Homa's implementation can achieve very similar tail slowdown to what we observed in the simulations. 99%-ile tail slowdown for short messages at 80% network load is 1.8–3.1 across a broad range of RPC sizes and workloads. We also studied the effect of priorities and controlled overcommitment in achieving low latency; removing priorities and controlled overcommitment from Homa can increase the tail slowdown of short messages by a factor 15.

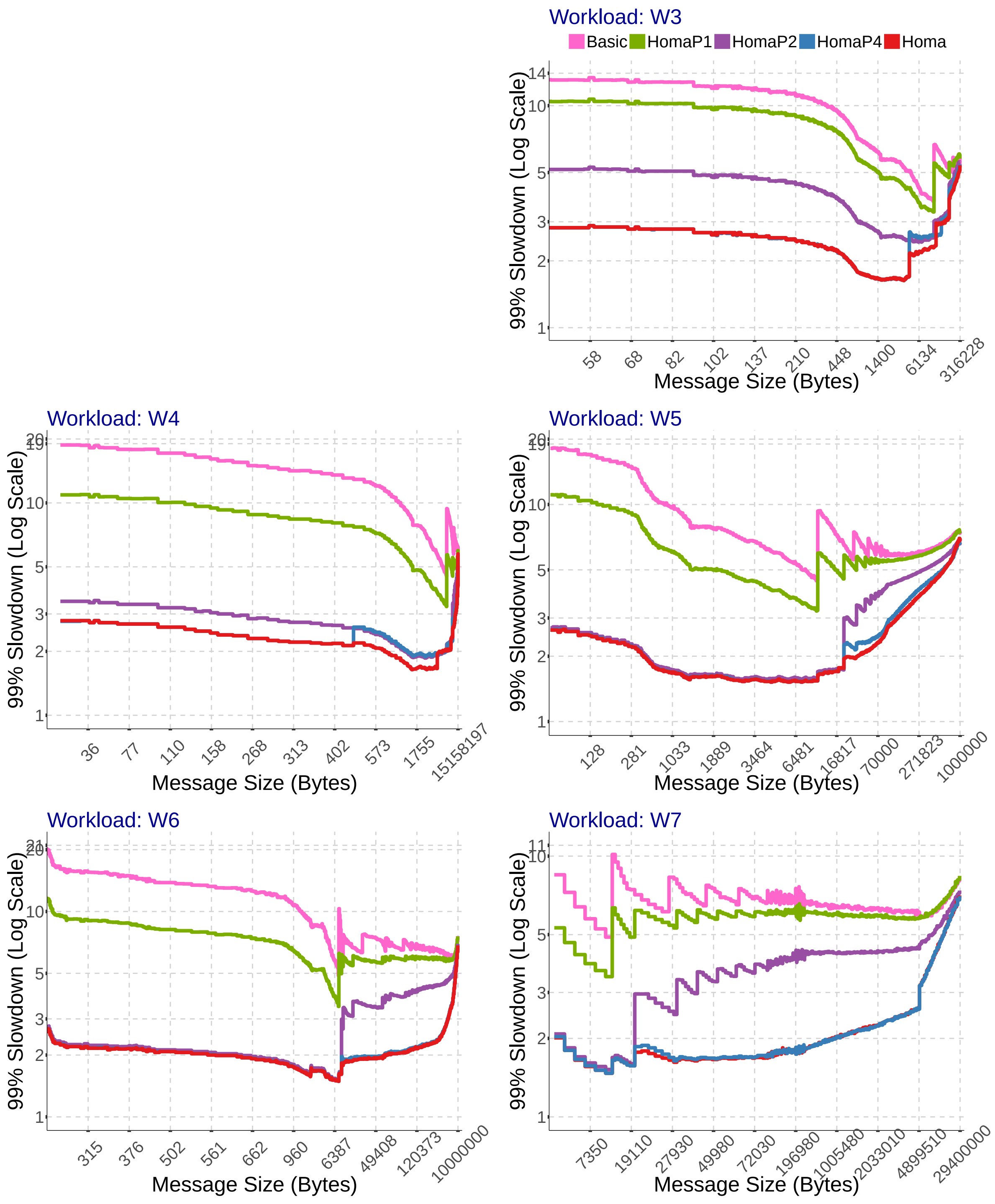


Figure 7.8: same as figure 7.3 except slowdown measured at 50% load

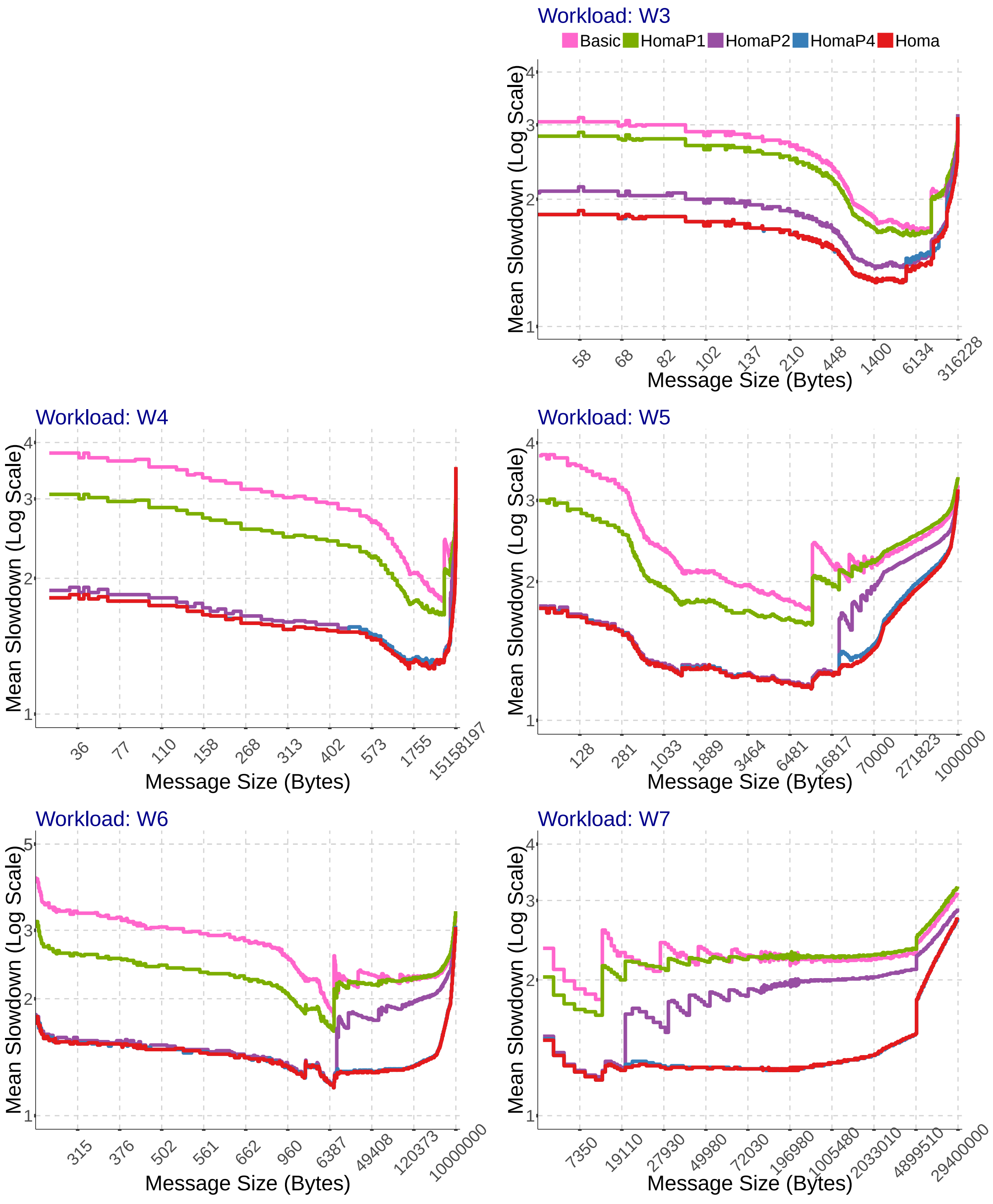


Figure 7.9: same as figure 7.3 except the y-axis is mean slowdown instead of 99th percentile slowdown and the experiments are run at 50% load.

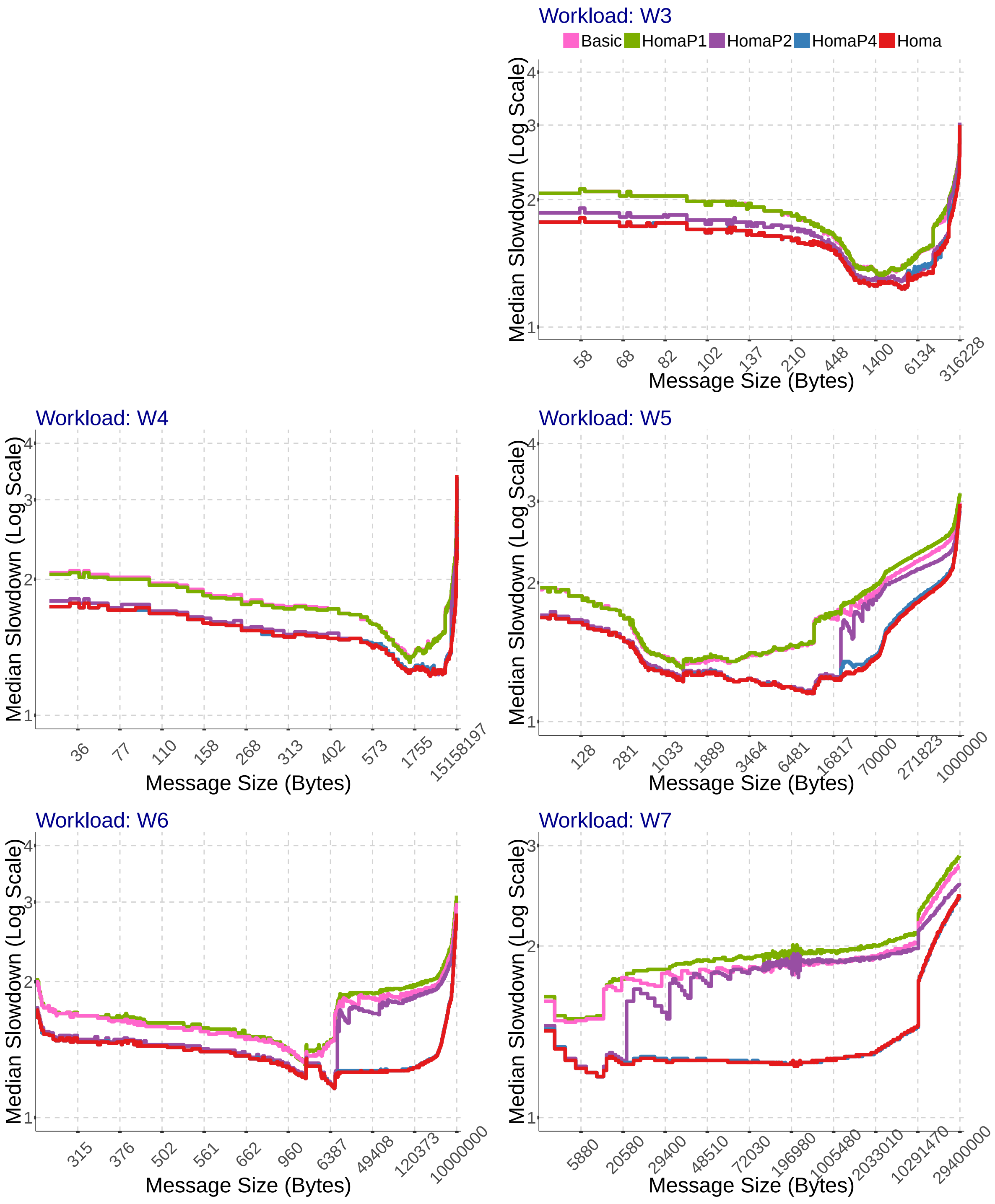


Figure 7.10: Same as Figure 7.3 except the y-axis is median slowdown instead of 99th percentile slowdown and the experiments are run at 50% load.

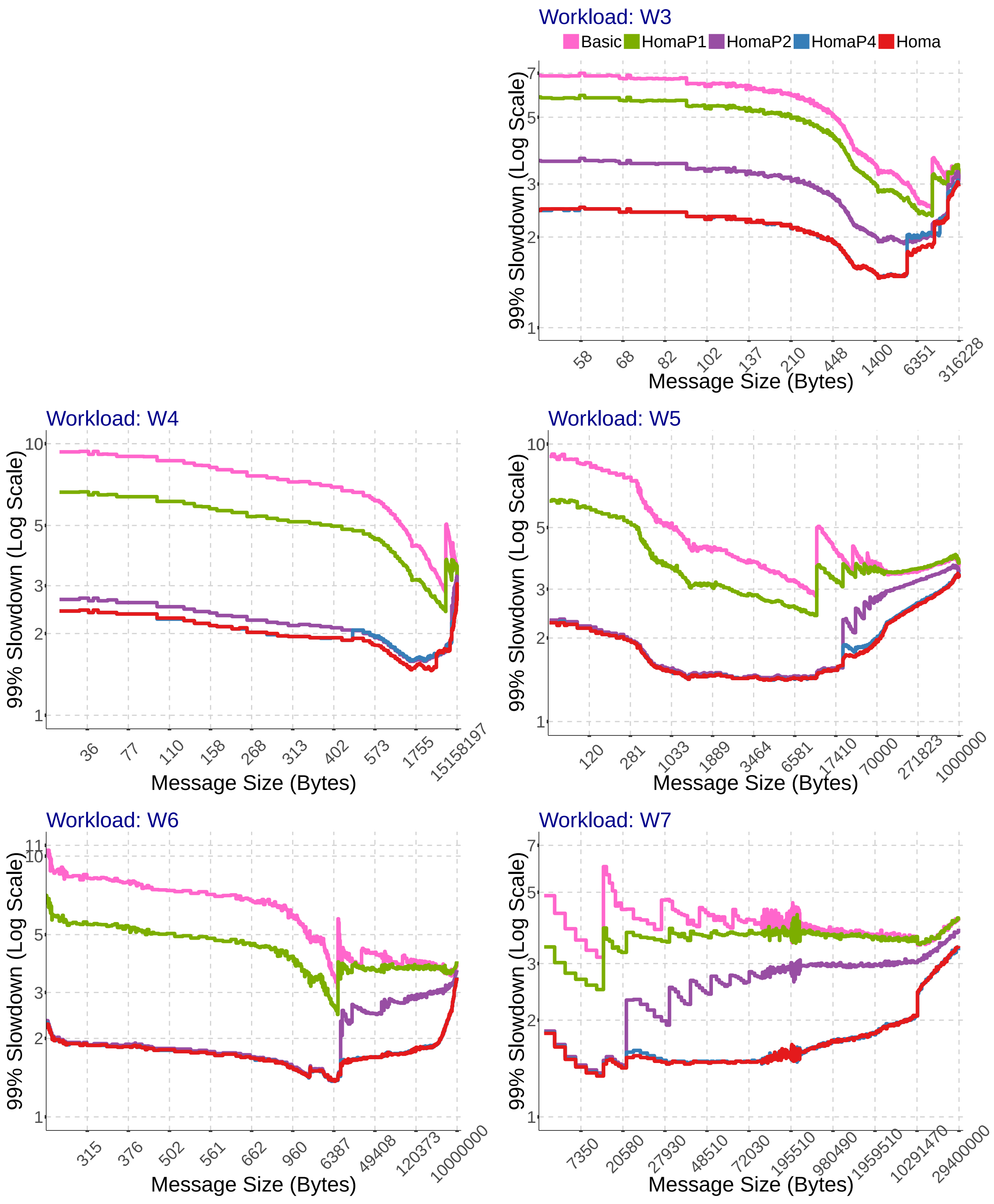


Figure 7.11: same as figure 7.3 except slowdown measured at 35% load

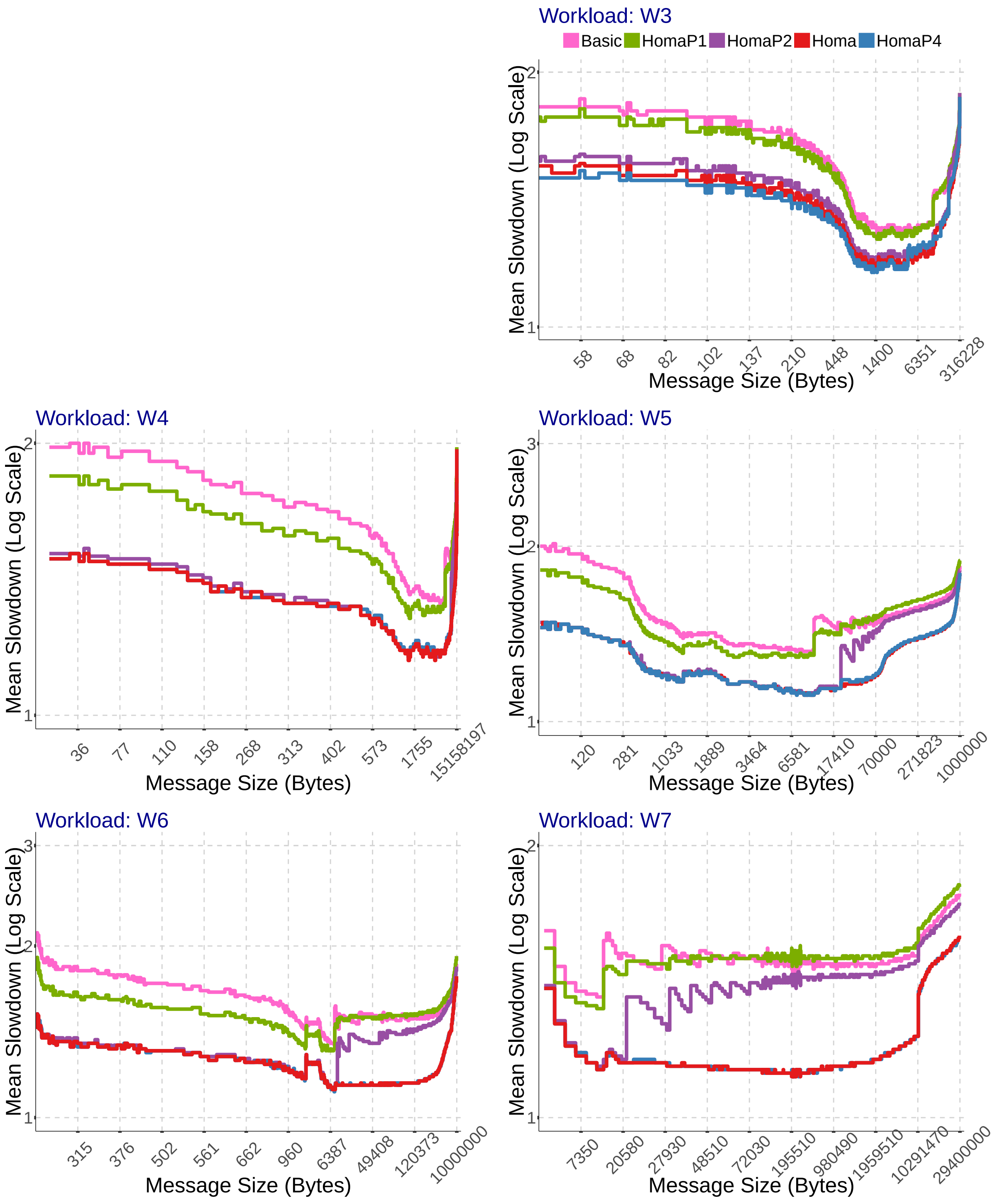


Figure 7.12: same as figure 7.3 except the y-axis is mean slowdown instead of 99th percentile slowdown and the experiments are run at 35% load.

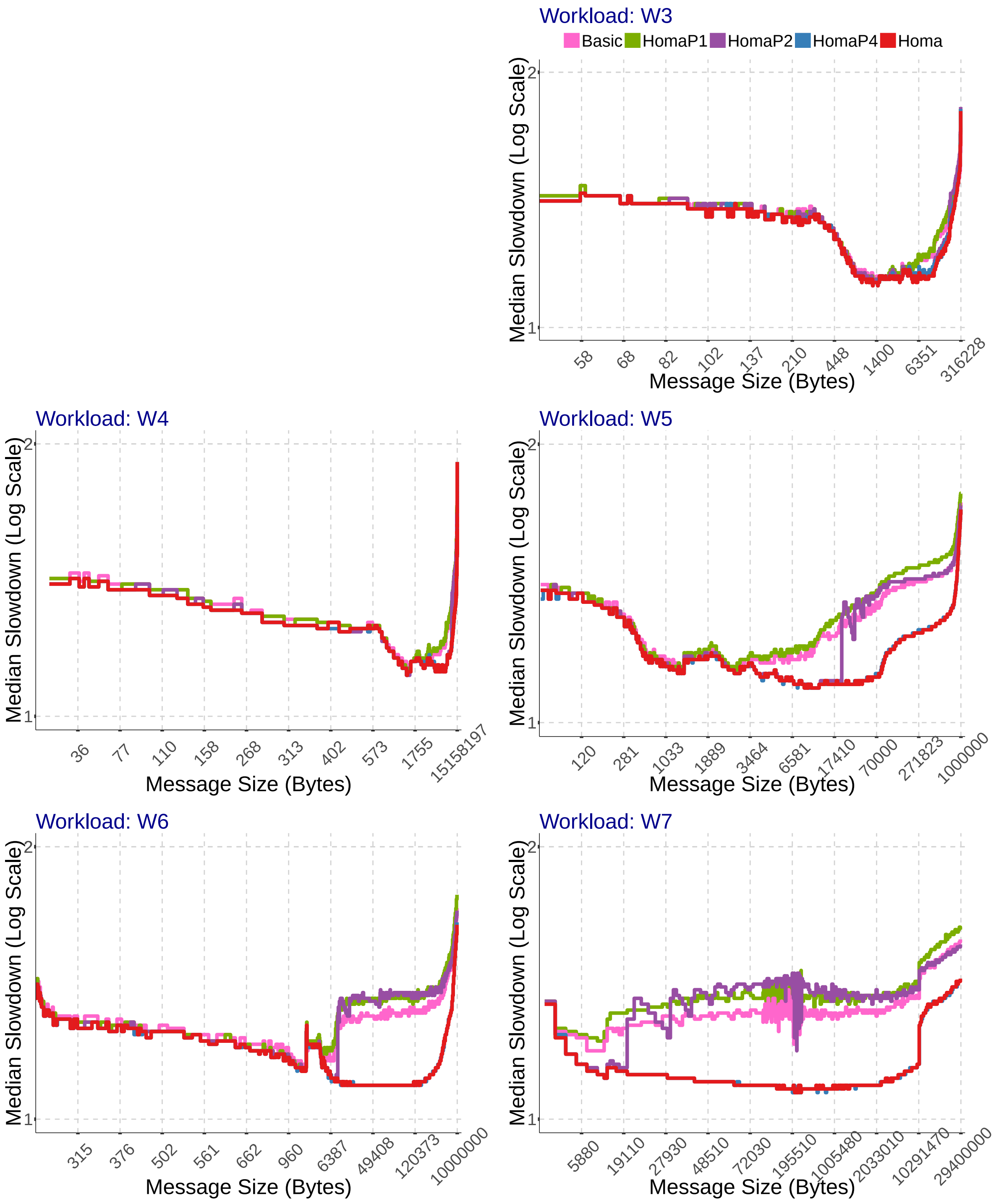


Figure 7.13: Same as Figure 7.3 except the y-axis is median slowdown instead of 99th percentile slowdown and the experiments are run at 35% load.

Chapter 8

Related Work

In recent years there have been numerous proposals for new transport protocols, driven by new datacenter applications and the well-documented shortcomings of TCP.

Some of these recent works like Hedera [1], MPTCP [29], and RPS [11] tackle the high throughput aspect of the design. Other works like HULL [4], PDQ [18], pHost [14], Fastpass [28], Timely [22], and DCQCN [38] tackle the low latency aspect of the design. However, none of these proposals combines the right set of features to produce low latency for short messages under high load.

Timely and DCQCN assume lossless networks for achieving low latency but the problem with these approaches is that they can't prevent pause frames and their negative impact in congestion spreading. The biggest problem with these approaches is that these works focus on low latency without considering the high throughput goal and performance of large flows.

The biggest shortcoming of most recent proposals is that they do not take advantage of in-network priority queues. This includes rate-control techniques such as DCTCP [3] and HULL [4], which reduce queue occupancy, and D^3 [36] and D^2TCP [35], which incorporate deadline-awareness. PDQ [18] adjusts flow rates to implement preemption, but its rate calculation is too slow for scheduling short messages. Without the use of priorities, none of these systems can achieve the rapid preemption needed by short messages.

A few systems have used in-network priorities, but they do not implement SRPT. Chapter 6 showed that the PIAS priority mechanism [7] performs worse than SRPT for most message sizes and workloads. QJUMP [15] requires priorities to be specified manually on a per-application basis. Karuna [8] uses priorities to separate deadline and non-deadline flows, and requires a global calculation for the non-deadline flows. Without receiver-driven SRPT, none of these systems can achieve low latency for short messages.

pFabric [5] implements SRPT by assuming fine-grained priority queues in network switches. Although this produces near-optimal latencies, it depends on features not available in existing switches.

pHost [14] and NDP [16] are the systems most similar to Homa, in that both use receiver-driven scheduling and priorities. pHost and NDP use only two priority levels with static assignment, which results in poor latency for short messages. Neither system uses overcommitment, which limits their ability to operate at high network load. NDP uses fair-share scheduling rather than SRPT, which results in high tail latencies. NDP includes an incast control mechanism, in which network switches drop all but the first few bytes of incoming packets when there is congestion. Homa's incast control mechanism achieves a similar effect using a software approach: instead of truncating packets in-flight (which wastes network bandwidth), senders are instructed by the protocol to limit how much data they send.

Almost all of the systems mentioned above, including DCTCP, pFabric, PIAS, and NDP, use a connection-oriented streaming approach. As previously discussed, this results in either high tail latency because of head-of-line blocking at senders, or an explosion of connections, which is impractical for large-scale datacenter applications.

A final alternative is to schedule all messages or packets for a cluster centrally, as in Fastpass [28]. However, communication with the central scheduler adds too much latency to provide good performance for short messages. In addition, scaling a system like Fastpass to a large cluster is challenging, particularly for workloads with many short messages.

Chapter 9

Limitations And Future Work

This section summarizes the most important assumptions Homa makes about its operating environment. If these assumptions are not met, then Homa may not achieve the performance levels reported here.

Homa assumes that congestion occurs primarily at host downlinks, not in the core of the network. Homa assumes per-packet spraying to ensure load balancing across core links, combined with sufficient overall capacity. Oversubscription is still possible, as long as there is enough aggregate bandwidth to avoid significant congestion. We hypothesize that congestion in the core of datacenter networks will be uncommon because it will not be cost-effective. If the core is congested, it will result in underutilization of servers, and the cost of this underutilization will likely exceed the cost of provisioning more core bandwidth. If the core does become congested, then Homa latencies will degrade. Homa's mechanisms for limiting buffer occupancy may reduce the impact of congestion in comparison to TCP-like protocols, but we leave a full exploration of this topic to future work.

Homa also assumes a single implementation of the protocol for each host-TOR link, such as in an operating system kernel running on bare hardware, so that Homa is aware of all incoming and outgoing traffic. If multiple independent Homa implementations share a single host-TOR link, they may make conflicting decisions. For example, each Homa implementation will independently overcommit the downlink and assign priorities based on the input traffic passing through that implementation. Multiple implementations can occur when a virtualized NIC is shared between multiple guest operating systems in a virtual machine environment, or between multiple applications that implement the protocol at user level. Obtaining good performance in these environments may require sharing state between the Homa implementations, perhaps by moving part of the protocol to the NIC or even the TOR. We leave an exploration of this problem and its potential solutions to future work.

Homa assumes that the most severe forms of incast are predictable because they are self-inflicted by outgoing RPCs; Homa handles these situations effectively. Unpredictable incasts can also occur, but Homa assumes that they are unlikely to have high degree. Homa can handle unpredictable incasts of several hundred messages with typical switch buffer capacities; unpredictable incasts larger than this will cause packet loss and degraded performance.

The Homa configuration and measurements in this paper were based on 10 Gbps link speeds. As link speeds increase in the future, RTTbytes will increase proportionally, and this will impact the protocol in several ways. A larger fraction of traffic will be sent unscheduled, so Homa's use of multiple priority levels for unscheduled packets will become more important. With faster networks, workloads will behave more like W1 and W4 in our measurements, rather than W5-W7. As RTTbytes increases, each message can potentially consume more space in switch buffers, and the degree of unpredictable incast that Homa can support will drop.

Homa is designed for use in datacenter networks and capitalizes on the properties of those networks; it is unlikely to work well in wide-area networks. There are many reason for why we believe Homa wont work in wide-area networks. One reason that may cause Homa to fail in these networks is that congestion can happen anywhere in the network and the assumption of Homa that congestion primarily happens at the TOR may not be valid anymore. Another reason is that these networks typically have very large and highly variable RTTs (RTTs in the range of 1–100ms is very common). This means lots of unscheduled packets in Homa that can easily cause buffer overflow and lots of packet drops. Homa's retransmission mechanisms is not optimized for an environment that packets drops are not rare.

Chapter 10

Conclusion

The rise of datacenter computing over the last decade has created new opportunities and challenges for network protocols. On one hand, modern datacenter networking hardware offers the potential for very low latency communication; round-trip times of 5 μs or less are now possible for short messages. In addition, many datacenter applications use request-response protocols that are dominated by very short messages (a few hundred bytes or less). Existing transport protocols are ill-suited to these conditions, so the latency they provide for short messages is far higher than the hardware potential, particularly under high network loads.

The combination of tiny messages and low-latency networks creates challenges and opportunities that have not been addressed by previous transport protocols. Therefore, in this dissertation we revisited the need for creating a low latency transport for datacenters. Homa meets this need with a new transport architecture that combines several unusual features:

1. It uses network priority queues with a hybrid allocation mechanism that approximates SRPT

2. It manages most of the protocol from the receiver, not the sender.

3. It overcommits receiver downlinks in order to maximize throughput at high network loads.

These features combine to produce nearly optimal latency for short messages across a variety of workloads. Even under high loads, tail latencies are within a small factor of the hardware limit. The remaining delays are almost entirely due to the absence of link-level packet preemption in current networks; there is little room for improvement in the protocol itself.

Homa can be implemented with no changes to networking hardware. We have implemented Homa in RAMCloud's transport layer and our implementation introduces new features that are different from traditional transports:

1. Homa implements discrete messages for remote procedure calls, not byte streams.
2. It is connectionless and has no explicit acknowledgments.

Homa's features significantly simplify the transport design and benefit the datacenter application. Our experiments show that our test bed implementation achieve very close results to what we observed in the simulations. We believe that Homa provides an attractive platform on which to build low-latency datacenter applications.

Bibliography

[1] AL-FARES, M., RADHAKRISHNAN, S., RAGHAVAN, B., HUANG, N., AND VAHDAT, A. Hedera: Dynamic Flow Scheduling for Data Center Networks. In Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation (Berkeley, CA, USA, 2010), NSDI'10, USENIX Association, pp. 19–19. 125

[2] ALIZADEH, M., EDSALL, T., DHARMAPURIKAR, S., VAIDYANATHAN, R., CHU, K., FINGERHUT, A., LAM, V. T., MATUS, F., PAN, R., YADAV, N., AND VARGHESE, G. CONGA: Distributed Congestion-aware Load Balancing for Datacenters. In Proceedings of the ACM SIGCOMM 2014 Conference (New York, NY, USA, 2014), SIGCOMM '14, ACM, pp. 503–514. 8, 18

[3] ALIZADEH, M., GREENBERG, A., MALTZ, D. A., PADHYE, J., PATEL, P., PRABHAKAR, B., SENGUPTA, S., AND SRIDHARAN, M. Data Center TCP (DCTCP). In Proceedings of the ACM SIGCOMM 2010 Conference (New York, NY, USA, 2010), SIGCOMM '10, ACM, pp. 63–74. 1, 5, 6, 7, 9, 46, 75, 125

[4] ALIZADEH, M., KABBANI, A., EDSALL, T., PRABHAKAR, B., VAHDAT, A., AND YASUDA, M. Less is More: Trading a Little Bandwidth for Ultra-low Latency in the Data Center. In Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (Berkeley, CA, USA, 2012), NSDI'12, USENIX Association, pp. 19–19. 1, 8, 9, 116, 125

[5] ALIZADEH, M., YANG, S., SHARIF, M., KATTI, S., MCKEOWN, N., PRABHAKAR, B., AND SHENKER, S. pFabric: Minimal Near-optimal Datacenter Transport. In Proceedings of the ACM SIGCOMM 2013 Conference (New York, NY, USA, 2013), SIGCOMM '13, ACM, pp. 435–446. v, 1, 3, 6, 7, 10, 73, 126

[6] ATIKOGLU, B., XU, Y., FRACHTENBERG, E., JIANG, S., AND PALECZNY, M. Workload Analysis of a Large-scale Key-value Store. In Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems (New York, NY, USA, 2012), SIGMETRICS '12, ACM, pp. 53–64. 7, 46

[7] BAI, W., CHEN, L., CHEN, K., HAN, D., TIAN, C., AND WANG, H. Information-agnostic Flow Scheduling for Commodity Data Centers. In Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation (Berkeley, CA, USA, 2015), NSDI'15, USENIX Association, pp. 455–468. v, 1, 3, 7, 10, 73, 116, 125

[8] CHEN, L., CHEN, K., BAI, W., AND ALIZADEH, M. Scheduling Mix-flows in Commodity Datacenters with Karuna. In Proceedings of the ACM SIGCOMM 2016 Conference (New York, NY, USA, 2016), SIGCOMM '16, ACM, pp. 174–187. 10, 125

[9] CHO, I., JANG, K., AND HAN, D. Credit-Scheduled Delay-Bounded Congestion Control for Datacenters. In Proceedings of the ACM SIGCOMM 2017 Conference (New York, NY, USA, 2017), SIGCOMM '17, ACM, pp. 239–252. 9

[10] Data Plane Development Kit. `http://dpdk.org/`. 5, 102

[11] DIXIT, A., PRAKASH, P., HU, Y. C., AND KOMPELLA, R. R. On the Impact of Packet Spraying in Data Center Networks. In Proceedings of IEEE Infocom (2013). 8, 18, 40, 125

[12] DRAGOJEVIĆ, A., NARAYANAN, D., CASTRO, M., AND HODSON, O. FaRM: Fast Remote Memory. In 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14) (Seattle, WA, Apr. 2014), USENIX Association, pp. 401–414. 8, 104

[13] FELDERMAN, B. Personal communication, February 2018. Google. 104

[14] GAO, P. X., NARAYAN, A., KUMAR, G., AGARWAL, R., RATNASAMY, S., AND SHENKER, S. pHost: Distributed Near-optimal Datacenter Transport over Commodity Network Fabric. In Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies (New York, NY, USA, 2015), CoNEXT '15, ACM, pp. 1:1–1:12. v, 3, 7, 10, 11, 73, 125, 126

[15] GROSVENOR, M. P., SCHWARZKOPF, M., GOG, I., WATSON, R. N. M., MOORE, A. W., HAND, S., AND CROWCROFT, J. Queues Don't Matter When You Can JUMP Them! In

12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15) (Oakland, CA, 2015), USENIX Association, pp. 1–14. 1, 10, 125

[16] HANDLEY, M., RAICIU, C., AGACHE, A., VOINESCU, A., MOORE, A. W., ANTICHIK, G., AND MOJCIK, M. Re-architecting Datacenter Networks and Stacks for Low Latency and High Performance. In Proceedings of the ACM SIGCOMM 2017 Conference (New York, NY, USA, 2017), SIGCOMM '17, ACM, pp. 29–42. v, 1, 3, 7, 8, 9, 12, 73, 84, 116, 126

[17] HE, K., ROZNER, E., AGARWAL, K., FELTER, W., CARTER, J., AND AKELLA, A. Presto: Edge-based Load Balancing for Fast Datacenter Networks. In Proceedings of the ACM SIGCOMM 2015 Conference (New York, NY, USA, 2015), SIGCOMM '15, ACM, pp. 465–478. 8, 18

[18] HONG, C.-Y., CAESAR, M., AND GODFREY, P. B. Finishing Flows Quickly with Preemptive Scheduling. In Proceedings of the ACM SIGCOMM 2012 Conference (New York, NY, USA, 2012), SIGCOMM '12, ACM, pp. 127–138. 1, 8, 10, 125

[19] JEONG, E., WOOD, S., JAMSHED, M., JEONG, H., IHM, S., HAN, D., AND PARK, K. mTCP: a Highly Scalable User-level TCP Stack for Multicore Systems. In 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14) (Seattle, WA, 2014), USENIX Association, pp. 489–502. 114

[20] LEE, C., PARK, S. J., KEJRIWAL, A., MATSUSHITA, S., AND OUSTERHOUT, J. Implementing Linearizability at Large Scale and Low Latency. In Proceedings of the 25th Symposium on Operating Systems Principles (New York, NY, USA, 2015), SOSP '15, ACM, pp. 71–86. 107

[21] memcached: a Distributed Memory Object Caching System. `http://www.memcached.org/`, Jan. 2011. iv, 1

[22] MITTAL, R., LAM, V. T., DUKKIPATI, N., BLEM, E., WASSEL, H., GHOBADI, M., VAHDAT, A., WANG, Y., WETHERALL, D., AND ZATS, D. TIMELY: RTT-based Congestion Control for the Datacenter. In Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication (New York, NY, USA, 2015), SIGCOMM '15, ACM, pp. 537–550. 5, 9, 125

[23] MONTAZERI, B., LI, Y., ALIZADEH, M., AND OUSTERHOUT, J. Homa: A Receiver-driven Low-latency Transport Protocol Using Network Priorities. In Proceedings of the 2018

Conference of the ACM Special Interest Group on Data Communication (New York, NY, USA, 2018), SIGCOMM '18, ACM, pp. 221–235. iv

[24] NISHTALA, R., FUGAL, H., GRIMM, S., KWIATKOWSKI, M., LEE, H., LI, H. C., MCELROY, R., PALECZNY, M., PEEK, D., SAAB, P., STAFFORD, D., TUNG, T., AND VENKATARAMANI, V. Scaling Memcache at Facebook. In 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13) (Lombard, IL, 2013), USENIX, pp. 385–398. 8, 104

[25] ns-2 Main Page. `http://nsnam.sourceforge.net/wiki/index.php/Main_Page`. 73

[26] OMNeT++ Discrete Event Simulator. `http://https://omnetpp.org/`. 73

[27] OUSTERHOUT, J., GOPALAN, A., GUPTA, A., KEJRIWAL, A., LEE, C., MONTAZERI, B., ONGARO, D., PARK, S. J., QIN, H., ROSENBLUM, M., ET AL. The RAMCloud Storage System. ACM Transactions on Computer Systems (TOCS) 33, 3 (2015), 7. iv, 1, 19, 102, 106

[28] PERRY, J., OUSTERHOUT, A., BALAKRISHNAN, H., SHAH, D., AND FUGAL, H. Fastpass: A Centralized "Zero-queue" Datacenter Network. In Proceedings of the ACM SIGCOMM 2014 Conference (New York, NY, USA, 2014), SIGCOMM '14, ACM, pp. 307–318. 1, 9, 88, 125, 126

[29] RAICIU, C., BARRE, S., PLUNTKE, C., GREENHALGH, A., WISCHIK, D., AND HANDLEY, M. Improving Datacenter Performance and Robustness with Multipath TCP. In Proceedings of the ACM SIGCOMM 2011 Conference (New York, NY, USA, 2011), SIGCOMM '11, ACM, pp. 266–277. 125

[30] Redis, Mar. 2015. http://redis.io. 1

[31] ROY, A., ZENG, H., BAGGA, J., PORTER, G., AND SNOEREN, A. C. Inside the Social Network's (Datacenter) Network. In Proceedings of the ACM SIGCOMM 2015 Conference (New York, NY, USA, 2015), SIGCOMM '15, ACM, pp. 123–137. 7, 46

[32] SHANLEY, T. Infiniband Network Architecture. Addison-Wesley Professional, 2003. 5

[33] SIVARAM, R. Some Measured Google Flow Sizes (2008). Google internal memo, available on request. 7, 46

[34] BCM56960 Series: High-Density 25/100 Gigabit Ethernet StrataXGS Tomahawk Ethernet Switch Series. `https://www.broadcom.com/products/ethernet-connectivity/switching/strataxgs/bcm56960-series`. 5

[35] VAMANAN, B., HASAN, J., AND VIJAYKUMAR, T. Deadline-aware Datacenter TCP (D2TCP). In Proceedings of the ACM SIGCOMM 2012 Conference (New York, NY, USA, 2012), SIGCOMM '12, ACM, pp. 115–126. 1, 125

[36] WILSON, C., BALLANI, H., KARAGIANNIS, T., AND ROWTRON, A. Better Never Than Late: Meeting Deadlines in Datacenter Networks. In Proceedings of the ACM SIGCOMM 2011 Conference (New York, NY, USA, 2011), SIGCOMM '11, ACM, pp. 50–61. 1, 125

[37] ZATS, D., DAS, T., MOHAN, P., BORTHAKUR, D., AND KATZ, R. DeTail: Reducing the Flow Completion Time Tail in Datacenter Networks. In Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (New York, NY, USA, 2012), SIGCOMM '12, ACM, pp. 139–150. 5

[38] ZHU, Y., ERAN, H., FIRESTONE, D., GUO, C., LIPSHTEYN, M., LIRON, Y., PADHYE, J., RAINDEL, S., YAHIA, M. H., AND ZHANG, M. Congestion Control for Large-Scale RDMA Deployments. In Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication (New York, NY, USA, 2015), SIGCOMM '15, ACM, pp. 523–536. 5, 9, 125