Leveraging Homa: Enhancing Datacenter RPC Transport Protocols

Xiaochun Lu, zijian Zhang

ByteDance
San Jose , United states
Xiaochun.lu@bytedance.com Zijianzhang@bytedance.com

Abstract

In hyper-scale data centers, the demand for transport protocols optimized for RPC performance is on the rise. Homa stands out as a protocol designed specifically for these settings, ensuring rapid request/reply messaging. Through our analysis, performance on Homa versus TCP revealed marked benefits, particularly for RPC messages below 50k, considering both latency and throughput. However, Homa's adoption as a universal RPC transport protocol in data centers faces following challenges:

- It is best suited for networks with hardware latencies under a few microseconds, restricting its use primarily to intra-rack servers.
- Homa's congestion window size is based on RTT_bytes (equivalent to BDP). Currently, a universal value is set for all peers, which doesn't align with the varied RTTs and receiver downlink bandwidths observed in data centers.
- Homa presumes a receiver's full downlink commitment, but real-world scenarios involve sharing with protocols like TCP, potentially leading to unnoticed congestion.
- Homa's message-based interface, while ensuring complete message delivery, hinders efficient pipelining, resulting in decreased throughput for larger RPC messages compared to TCP.

This paper introduces enhancements to Homa's congestion control, incorporating dynamic RTT detection and peer adjustable windows based on RTT-induced congestion insights. These improvements facilitate Homa's applicability in environments with non-uniform RTTs and enable its coexistence with TCP traffic without compromising throughput.

Keywords

Data Centers; RPC; Low latency; Transport protocols; Throughput; Congestion Control;

Introduction

Today, hyperscale Data center networks are becoming increasingly complex, with a variety of different devices and technologies being used. This complexity can make it difficult to achieve low latency and high throughput in RPC communication. It also adds challenges to adopting new technology.

Currently, TCP is the primary transport protocol for RPC communication in these data centers, owing to its reliability, end-to-end error management, and ubiquitous support across major OSs and programming languages. However, TCP's high latency for short messages, particularly under mixed workloads, underscores the need for innovative transport protocols and advanced congestion control mechanisms.

RDMA, a high-performance network technology, promises to trim down RPC communication latency. It facilitates applications in accessing memory across networked machines without engaging the operating system, thereby considerably reducing latency, especially for short messages. Yet, the complexities in programming, memory management, and the prerequisite of RDMA-equipped NICs and switches, along with its inherent scalability limitation concerning concurrent connections, can sometimes overshadow its advantages.

Enter Homa [1]– an innovative solution tailored for RPC frameworks in data centers. It boasts unparalleled efficiency for networks with microsecond-range hardware latencies, striving for the most minimal tail latency for short messages, even amidst high network loads and varied message lengths.

Our performance analysis of Homa against TCP, highlighting its clear edge, especially for RPC messages under 50k, regarding latency and throughput. Yet, for message sizes exceeding 50k, Homa's throughput efficacy is inferior to that of TCP.

Currently, Homa assuming network is uniform and a single message will consume the entire downlink bandwidth of a receiver. The transmission and reception window sizes are guided by RTT_bytes, equating to the product of a network link's capacity and its round-trip delay. Fine-tuning this value is crucial for Homa's optimal performance. Presently, a single pre-set value is applied to all peers, but this isn't ideal given the diverse RTTs and receiver downlink bandwidths. Real-time per-peer RTT detection is essential.

Additionally, Homa permits senders to unilaterally transmit one BDP of data for each message without awaiting grants. If multiple large messages are ready to be sent to the same recipient (multiple nodes incast with large message), congestion issues arise . Current Homa lacks congestion control in these situations. When the buffered unscheduled packets exceed thresholds, Homa should proactively send alerts to sender to curb the limits for new transmissions.

Another oversight is that Homa assumes unscheduled packets will dominate the sender-side bandwidth. With predominantly large messages, the scheduled section could consume significant bandwidth, necessitating a decrease in unscheduled data allocation. Receivers can send unscheduled packet ratio to sender side to adjust allocations of unscheduled bytes based on recent traffic patterns.

Homa assumes that the entire downlink of receiver is dedicated to homa. But in reality, the downlink mostly needs to be shared with other protocols like TCP. This will introduce congestion that is not easily detected by homa. Homa needs capability to detect congestion due to network resources shared with other protocols and dynamically adjust unscheduled traffic based on congestion level.
This paper delves deep into augmenting the throughput performance of extensive RPC messages and focuses on enhancing congestion control in Homa to extend its usages.

We propose several enhancements to Homa in this paper:

1. Dynamic per-peer BDP detection.
2. RTT-informed congestion detection.
3. Adaptive per-peer window adjustments for congestion management.
4. Introduce "Homa RPC streaming".

Homa Congestion Control Enhancements

per-peer RTT detection

The BDP should be computed using the minimal RTT (Round-Trip Time) value. Employing this approach aids in mitigating latency perturbations instigated by the accumulation in software buffers. This minimum RTT, referred to as peer_rtt_min, is discovered via the RTT_PROBE control message.

Homa will continuously monitor the RTT value for every active peer using the RTT_PROBE control message. This message is dispatched whenever there's a new message coming in, and the interval exceeds 1 ms. The priority for this RTT_PROBE is aligned with that of the briefest unscheduled packet, ensuring its direct transmission without any throttling.

The RTT probe interval can be adjusted through sysctl value rtt_probe_interval_us. It's default value is set to 1ms to reduce the overhead . When the recent RTT value is above the certain high threshold, network congestion is detected, the probe interval will be changed more frequently to keep recent RTT value more updated. Currently, this new interval is set to 8 * RTT_MIN when network congestion is detected in the peer down link.

RTT_PROBE message is used to detect each peer's current RTT value, called peer_rtt_recent. Here is the definition of rtt_probe and rtt_response message

```
struct rtt_probe_header {
struct common_header common;
__be64 timestamp_ns;
__be32 link_mbps;
} __attribute__((packed));
```

When peer receives PROBE message, it will respond with RTT_RESPS control message directly. RTT_RESPS is defined with following structure format

```
struct rtt_probe_resp_header {
struct common_header common;
__be64 timestamp_ns;
__be32 link_mbps;
} __attribute__((packed));
```

When sender receives RTT_RESPS message, it will compare it with current timestamp to calculate peer's current RTT called peer_rtt_recent. peer_rtt_min defines the minimum value received among all peer_rtt_recent.
If detected peer_rtt_recent is less than peer_rtt_min, peer_rtt_min will be updated. After a few rounds of probing , peer_rtt_min will be stabilized. Peer BDP value will be calculated based on peer_rtt_min.

To reduce noise, peer_rtt_recent can be averaged with previous value.

Congestion detection

Homa uses peer_rtt_recent and peer_rtt_min to detect traffic jams for the receiver.
There are a few peer_rtt_min based threshold values defined to measure traffic condition of the link:
gso_delay = gso_pkt_data_size * 8 / peer_link_mbps ;
rtt_low = 2*(peer_rtt_min + gso_delay);
rtt_mid = 2 * rtt_low;
rtt_high = 8* peer_rtt_min;
gso_delay is value for how long a gso batch packets can be transmitted to wire. This delay is added for all data packets, but control message is sent directly without batch. This value is added back to rtt_min when calculating the other rtt threshold.
When peer_rtt_recent > rtt_high, congestion is detected. The sender should use low limit of unscheduled packet (calculated using rtt_min) to send RPC data packet .
When peer_rtt_recent < rtt_low, the link is idle, sender can send more unscheduled packet.
To support diverse network environments, Peer_link_mbps is also passed to peer via RTT_PROBE_RESPONSE

message. along with unscheduled_ratio defined in the below section.

Unscheduled ratio

Initially, the receiver determines the percentage of unscheduled bytes in the total incoming bytes, termed as the "unscheduled ratio". This ratio is allocated the topmost priorities for unscheduled packets, with the remaining priorities set aside for scheduled packets. The division of unscheduled priorities is structured so that each tier manages a consistent byte amount, and shorter messages are accorded higher priorities. This unscheduled ratio is then communicated to the sender via grant message.

.

This ratio reflects the traffic pattern in the near future, since the ratio reflects data collected from the first packet of new messages.

The unscheduled_ratio is also sent to peer through RTT_PROBE message in case there are no grants to send to the sender.

Unscheduled window (RTT_bytes)

Homa adjusts how many "unscheduled" packets it sends for each new message based on recent network activity. Here's how it decides how many unscheduled packets to send:

If the current delay (peer_rtt_recent) is lower than rtt_mid , Homa thinks the network is clear. So, it sends more unscheduled packets by setting rtt_unscheduled to rtt_mid

If the current delay is in the range between rtt_mid and rtt_high , it enters congestion control area, Homa will send out less unscheduled bytes. The formula to calculate rtt_unscheduled is:

rtt_unscheduled=rtt_low - (peer_rtt_recent - peer_rtt_mid) *3 /8;

If the current delay is above rtt_high, Homa believes there's a traffic jam on the network. To avoid adding to the jam, it aggressively reduces the number of unscheduled packets by setting rtt_unschduled to one GSO size.

Here is the overall algorithm to determine the rtt_unschdule value:

```
SET rtt_unscheduled TO peer_rtt_recent
IF rtt_unscheduled IS LESS THAN rtt_mid THEN
    SET rtt_unscheduled TO rtt_mid
ELSE IF rtt_unscheduled IS GREATER THAN rtt_high THEN
    SET rtt_unscheduled TO 1
ELSE IF rtt_unscheduled IS GREATER THAN rtt_mid AND rtt_unscheduled IS LESS THAN rtt_high THEN
    SET rtt_unscheduled TO rtt_low - (peer_rtt_recent - peer_rtt_mid) *3 /8
END IF
```

unscheduled_bytes = rtt_unscheduled * peer_link_mbps /8

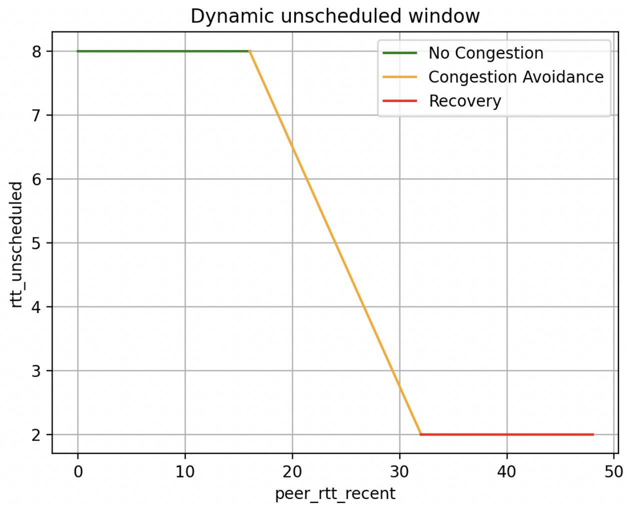

Figure 1 Dynamic adjustable unscheduled window

Most of time, unscheduled_bytes is calculated using rtt_mid, which is stable value calculated using rtt_min.

Another observation is that it takes about rtt_mid time from sending first packets to get grants. There is a gap between receivers consuming all unscheduled packets and sender able to send out scheduled bytes. To fill this gap, the sender can check whether the RPC throttle list is empty or not. If it is empty, it can send more unscheduled bytes to fill this gap. The formula for calculating RTTbytes will be :

Figure 1 Dynamic Adjustable Unscheduled Window

```
IF RPC throttle list is empty
    set rtt_unscheduled to rtt_mid
    IF unscheduled_ratio is less than 40 %
        Set rtt_unscheduled to half of current value
    ENDIF
ENDIF
```

Scheduled Window

The scheduled window on the receiver side is designated for allocating grants to peers. It restricts the allocation to one grant per message for each peer, ensuring a continuous flow of BDP packets in the pipeline. The formula for the scheduled window is: scheduled_window = rtt_mid *local_link_mbps >> 3;

peer_rtt_recent is not used here due to asymmetric nature. rtt_recent detected at receiver side may not equal to send side. Instead, rtt_mid is used to keep scheduled BDP packets at a more constant rate to reduce burst.

In this context, the scheduled ratio isn't considered here, since Homa employs overcommitment to enhance the pipeline's efficiency. When dealing with smaller grants,

scheduled packets can consume significant processing cycles. Thus, allocating larger grants can optimize the transmission of current priority messages. Typically, each grant should be rounded up to a single GRO to further enhance throughput.
One possible enhancement is skip grant to specific peer that has peer_rtt_recent is bigger than rtt_high. We will leave this for future exploration.

Homa Bidirectional RPC streaming Enhancement

Since the Homa RPCs in the same stream share a lot of the same attributes, we don't need to create a brand new Homa RPC for each request/resposne in the stream. The overhead of memory management for ***struct sk_buffs*** can not be saved, but we can save some efforts for locking logic and the management of ***struct homa_rpcs*** themselves.

When the number of ongoing Homa RPCs is not high, the overhead of locking and ***struct homa_rpcs*** can be ignored. However, under the heavy traffic load, when there are a lot of ongoing Homa RPCs, frequently creating and reclaiming Homa RPCs can incur unfavorable overhead and exacerbate lock contention.

In order to solve this problem, we equip the Homa Module with the ability to support stream RPC internally. All of the RPCs in the same stream can reuse one Homa RPC. Decided by users, they can terminate the stream at any time. As a result, Homa works more robustly and efficiently when there are multiple ongoing stream RPCs.
On the other hand, since stream RPC is not internally supported by the Homa module, grpc-homa takes some efforts to implement it at the application level. With streamed Homa RPCs, applications can support stream-based operation easier by just mapping stream ID to Homa RPC ID.

Test Setup

25G network hardware:

CPU: Intel(R) Xeon(R) Platinum 8163 (96 core,2.50GHz) RAM: 400G DIMM DDR4 NIC: Mellanox ConnectX–4 Lx 25 Gbp TOR Switch; Arista DCS-7050SX3-48YC12-F 25G ports

100G network hardware:

CPU: Intel(R) Xeon(R) Silver 4314 (64 cores, 2.4 GHz) RAM: 400 GB DIMM DDR4 NIC: Mellanox Technologies MT28841 dual-port 100Gb/s

TOR Switch: Ruijie Networks RG-S6580-48CQ8QC 100G ports

Software

Debian 10 VM s are running on each host to run cluster benchmark test. Each vm is running Linux 5.15 kernel with homa modules loaded. One Mellanox VF is assigned to each VM using SR-IOV. Each VM has assigned 8 vCPU and 16 G memory

Test Tool

cp_node is a program to test the performance(including throughput, latency, etc) of Homa or TCP.

In our test, we mainly tweak some parameters for clients to adjust the behavior of the client node.

- workload, workload to run the test, could be fixed-size or workload type.
- client-max, maximum number of outstanding requests from a single client machine (divided equally among client ports).
- ports, number of ports on which to send requests (one sending/receiving thread per port).

On the other hand, for both Homa server and TCP server, we have one thread to handle all of the incoming requests. For fixed-size requests, servers reply with a 100 bytes response. For workload W4 and W5, servers reply with the same length of the request.

Performance Evaluation

Basic performance evaluation

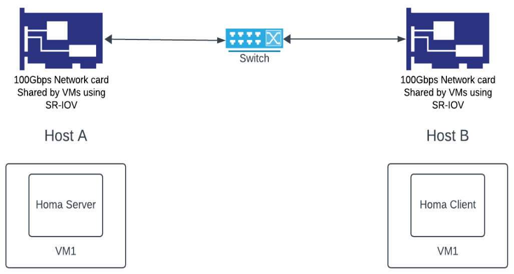
Switch
100Gbps Network card Shared by VMs using SR-IOV
Host A
100Gbps Network card Shared by VMs using SR-IOV
Host B
Homa Server
VM1
Homa Client
VM1

Figure 2 Basic performance test setup

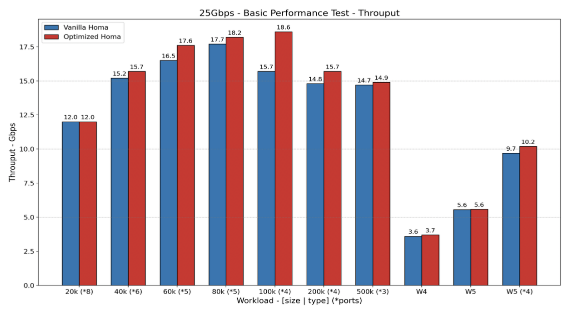

Figure 3 Basic Performance Test-Throughput Result for 25Gbps network

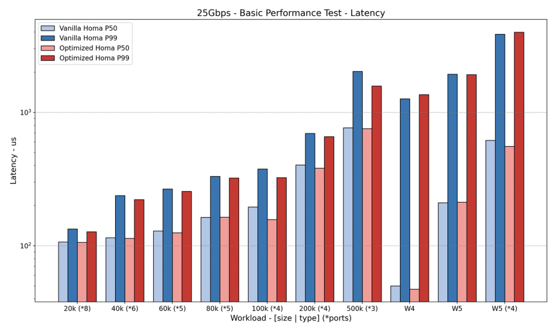

Figure 4 Basic Performance Test-latency Result for 25Gbps network

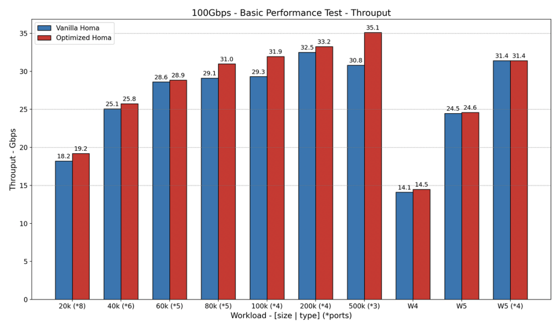

Figure 5 Basic Performance Test-Throughput Result for 100 Gbps network

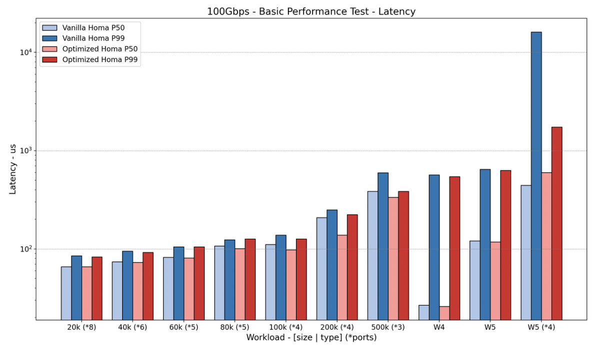

Figure 6 Basic Performance Test-latency Result for 100 Gbps network

In the experimental findings(Figure3-6), the optimized algorithm demonstrates 5% to 14% throughput improvement for large size message between 40k to 500k compared to static -Cast RTT_bytes configurations under favorable network conditions (utilizing less than 50% of network capacity). This substantiates the precision of the RTT detection mechanism. Additionally, under the w5 workload with four concurrent RPC requests, the standard Homa protocol encounters buffer overflow issues, whereas the enhanced Homa successfully circumvents them.

In-cast Test

In the in-cast evaluation involving large messages, an experiment was conducted between a Homa server operating on Host A and six Homa client virtual machines situated on Host B. The setup illustrated by figure 7.

Figure 8 show the latency test results under in-cast for 100 Gbps network.

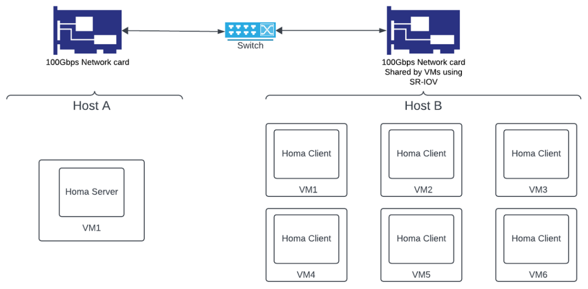

Figure 7 testbed setup for in-result cast test

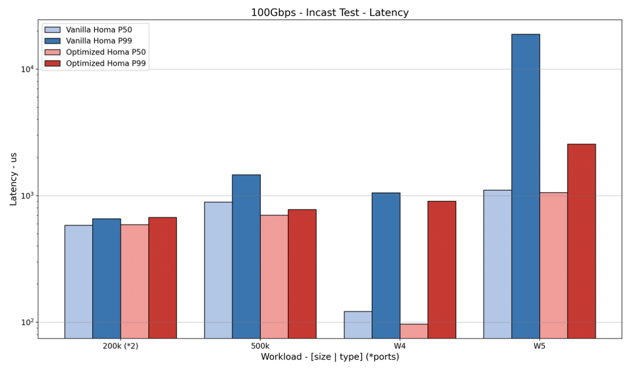

Figure 8 In-cast Test Latency Result

Vanilla Homa, utilizing static RTTbytes, fails to proactively manage the in-cast scenario when multiple servers concurrently direct requests to a singular server. Notably, under the w5 workload conditions, the Vanilla Homa protocol experienced buffer overflow. In contrast, the augmented Homa variant effectively mitigated these challenges. This underscores the efficacy of the integrated congestion detection mechanisms and adaptive window adjustments in minimizing in-cast-related congestion.

Bandwidth Split Test

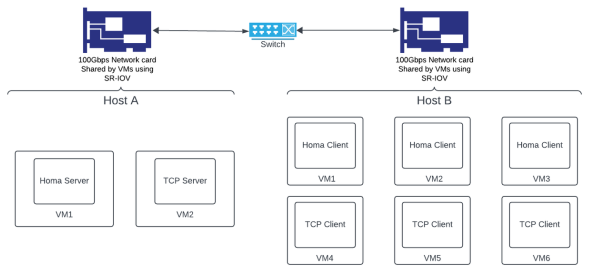

Figure 9 split traffic test setup

In bandwidth split test illustrated by figure 9, a Homa server and a TCP server operating on Host A, and three Homa client virtual machines and three TCP virtual machines situated on Host B.

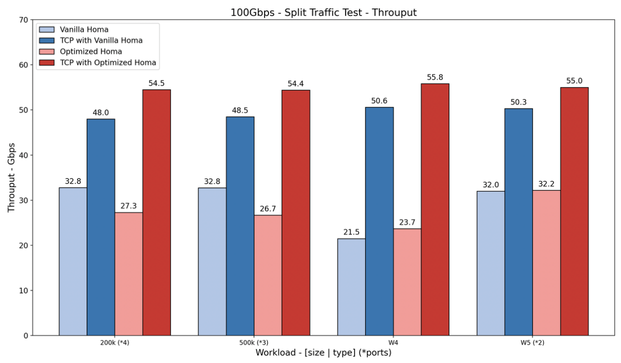

Figure 10 Split Traffic Throughput Test Result for 100Gbps Network

To introduce under loader network conditions, TCP traffic is generating using W5 load , which will consume 50 G bandwidth, which about half of the total avaible bandwidth. Then, adding Homa traffic between the Homa server and three Homa client nodes to observe the impact of sharing Homa traffic and TCP traffic on the same NIC and Switch ports. Figure 10 and 11 presents latency and throughput metrics for both the refined and standard Homa implementations when sharing traffic with TCP.

Based on the empirical observations, Homa traffic demonstrates harmonious coexistence with TCP. There is minimal mutual throughput interference when ample bandwidth is available for both traffic types. The refined algorithm adeptly identifies network congestion instigated by TCP, subsequently regulating its traffic emission to the network. It is pivotal to note that the enhanced algorithm adeptly circumvents buffer overflow challenges under the W5 workload, whereas the traditional Homa protocol manifests significant latency due to this anomaly.

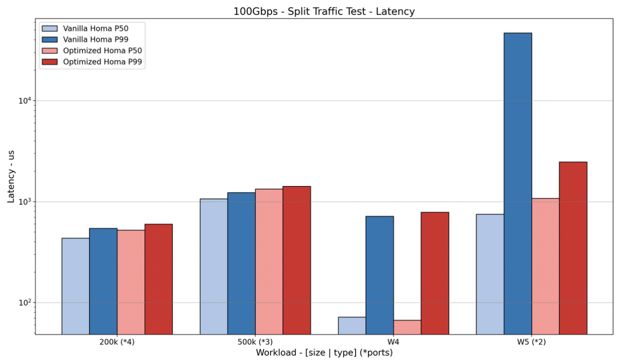

Figure 11 *Split Traffic Latency Test Result for 100Gbps Network*

Future work

More accurate RTT measurement: More accurate measurement requires distinguishing between fabric (network) delay and host software delay in RTT. Precise time stamping can be applied at the very edge of the host's network interface card (NIC) for both transmitted and received RTT probe packet.

Congestion detection base on average RTT deviation: Deviation of average RTT (rtt_avg) from RTT_mid (three times RTT_min) can be used for congestion detection.

Optimize pacer: Pacer needs to be aware of network congestion based on recent RTT. When RTT is low, although GRANT is not received, try to send some packets at a relatively low rate. When RTT is high, transmit less packets to NIC.

Conclusion

In conclusion, the developed RTT detection algorithm demonstrates proficiency in identifying the minimum RTT for a link, subsequently employing this as a foundation to define the optimal RTTbytes for unscheduled packets. Experimental results demonstrate 5% to 14% throughput improvement for large size messages between 40k to 500k compared to manual RTTbytes configurations. Notably, within heterogeneously structured networks, this algorithm adeptly detects near-optimal unscheduled windows (RTTbytes) tailored to individual links characterized by diverse latencies and bandwidth capacities.

Moreover, the novel algorithm exhibits resilience against buffer overflow scenarios precipitated by big messages in incast situations, particularly when operating with big unscheduled bytes. Our investigations further reveal Homa's capability to harmoniously coexist with TCP traffic. The protocol adeptly recognizes congestion instigated by TCP traffic, judiciously moderating its packet transmissions to the network. During incast evaluations, Homa nodes consistently demonstrated equitable downlink sharing, facilitated by receiver-side congestion control mechanisms.

References

Proceedings Paper Published
1. John Ousterhout Stanford University, “A Linux Kernel Implementation of the Homa Transport Protocol”, 2021 USENIX Annual Technical Conference.
2. Radhika Mittal∗ (UC Berkeley), Vinh The Lam, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi∗ (Microsoft), Amin Vahdat, Yaogong Wang, David Wetherall, David Zats, “TIMELY: RTT-based Congestion Control for the Datacenter” SIGCOMM ’15 August 17-21, 2015, London, United Kingdom
3. Behnam Montazeri, Yilong Li, Mohammad Alizadeh† , and John Ousterhout Stanford University, +MIT, “Homa: A Receiver-Driven Low-Latency Transport Protocol Using Network Priorities”,SIGCOMM ’18, August 20-25, 2018, Budapest, Hungary

Acknowledgements

We would like to extend our gratitude to John Ousterhout from Stanford University for his invaluable advice and direction. This work was supported by bytedance.