A Practical Guide To Reliable UDP

Shane Miller (gshanemiller6@gmail.com)

2023-02-25

Abstract

This paper provides a practical guide to implementing reliable UDP packet processing complementing larger engineering efforts in distributed computing or RPC (remote procedure call). Rather than new theory, this paper builds on the excellent work in [9, 4, 8, 11] by emphasizing code implementation. Reliability must resolve the system issue of congestion, and per packet details of preparation and loss detection. Congestion control determines when to send data which is connected to how fast data is sent. Packet management supplements congestion measures providing low level packet pacing, and feedback signals used by the congestion algorithm. When composed with kernel bypass network libraries, it's possible to complete "up to 10 million small RPCs per second, or send large messages at 75 Gbps" on one CPU core [8]. This work is applicable for commercial corporate networks within data centers excluding WAN. No speciality hardware is required.

1 Introduction

Research on network protocols for data movement across computer networks is a mature with broad connections to compression (entropy), hardware design, serialization, and networking library choice ranging from system calls to full operating system by-pass.

Within the space of HPC (high performance computing) efficiently sharing state is paramount. HPC sends UDP packets over speciality lossless hardware. Mellanox figures prominently here. Speciality hardware can include FPGA assist. Various forms of RDMA (remote direct memory access) can offload memory access from the CPU onto NICs delivering even higher performance. Solarflare, a NIC manufacturer often used in the finance domain, sells offload libraries making UNIX socket systems calls run fast. See [10, 6, 7, 2, 5, 12].

All these solutions require premium CAPEX (capital expenditure) in hardware, software licensing, or extensive system administration work. [12] depends all participating hosts synchronizing clocks to within a "few hundred nanoseconds." They are also exclusionary. The more networking details move away from UDP, the harder it is for the system to interoperate with commercial applications.

In this paper, we eschew high cost speciality solutions focusing instead on practical design for commercial corporate networks where packets move within data centers. Data movement between data centers or WAN (wide area networks) is out of scope. There's too much overhead in packet forwarding. See section 6.

This work is based on `ePRC` [11] which describes a "general-purpose RPC library" that's competitive with "specialized systems." It can complete "up to 10 million small RPCs per second" on one CPU core.

Achieving low-latency high-bandwidth data movement is a system concern. Answering it requires a multi-scale analysis at all levels of software design from architecture to packet design. `ePRC`'s solution composes several strategies into an integrated whole consisting of six parts.

First, packets are sent or received through kernel by-pass libraries. DPDK [1] is the principle example here. It was written by a consortium of companies including Intel, Mellanox, AMD, Microsoft, and

others. Rather than using the operating system to manage packet movement, by-pass libraries communicate directly with the host's NIC through polling. Although programmer's are tasked with making and managing individual packets, DPDK can outperform the kernel by an order of magnitude or more. DPDK supports all IP protocols and a wide range of NICs including Mellanox, and Solarflare.

Second, `ePRC` uses UDP to move data. Unlike TCP, it is not encumbered by protocols to manage transmission rate or guarantee packet delivery in order. It scales better than RDMA because client/servers can run more connected sessions per host.

Third, `ePRC` uses Timely congestion [8] to calculate data transmission rates. Timely uses per packet RTT (round trip time) signals to decrease sending rates when congestion is high, and increase rates when RTTs are low. RTTs are calculated in code, or may use NIC timestamps if supported. The only requirement is accuracy. See section 7.

Next, `ePRC` uses Carousel [9] in certain cases to temporally delay packet transmission so that outgoing packets flow at the rate calculated by Timely.

Timely and Carousel solve congestion. In widely deployed applications suites, multiple hosts are simultaneously transceiving data. If the hosts share a common switch or router it's possible for packets to be delayed or lost by stressing the component's processing capacity. RTTs will increase. In these cases, all individual hosts can do is change its transmission rate until congestion resolves.

Next, `ePRC` carefully designs its server threads to coordinate well with request intake and response processing. It rightly describes it as "a key design decision." Any viable solution must prevent head-of-line blocking. Delegation of requests to worker threads can help, but not if it requires expensive hand-offs through data copying or synchronization.

Finally, `ePRC` optimizes data flow by not assuming the worst case. When packets are kept small and RTTs are managed, congestion overhead may be skipped. In section 5.2.2 `ePRC` reports "99% of all datacenter links are less than 10% utilized at one-minute timescales" and "90% of top-of-rack switch links, which are the most congested switches, are less than 10% utilized at 25 µs timescales." Consequently, congestion overhead is not indicated for every packet.

2 Organization

This papers works in a spiral pattern circling from higher levels of abstraction starting with congestion into lower level concerns like session management. To avoid excessive detail in any one section, we periodically pause to summarize or integrate salient details omitted earlier. Source code is provided in `GITHUB` in three places. Congestion research, including this document, can be here. Code implementing this design is located here. `eRPC` code is found here.

Overall, the paper is organized in four major pieces. Congestion is engaged first starting with Timely. Timely calculates the rate at which packet $n + 1$ should be sent based on the RTT for the last packet n. Timely works with Carousel to send packets at prescribed future time to consummate the calculated rate. This work also covers when congestion control should be used, and when it can be avoided.

The second piece deals with design. The salient concerns are sever thread orchestration, session management, packet pacing, and packet memory cleanup. Like `eRPC` we desire a system where requests can be handled in place at the receiver or delegated to worker threads without undo overhead.

Third, we address low level implementation details such as creating packets in DPDK, methods for calculating RTTs, and avoiding running CPUs at 100% duty.

Finally, the design is benchmarked through a simple KV (key-value) system. (TBD: add benchmark and describe system).

3 Timely Motivation

Timely [8] section 4.2 envisions N end hosts all sending data simultaneously with a combined rate $y(t)$ bytes/sec. The N transmitters share a congestion point through which packets must traverse. Suppose this congestion point empties its queue at rate C bytes/sec.

Now, if at some time t, we have $y(t) > C$ congestion

becomes worse. It's getting more input data per unit time than it can process described by $y(t)-C$.Timely posits a dimensionless gradient in terms of a queuing delay $q(t)$ as:

$$\frac{dq(t)}{dt} = \frac{y(t) - C}{C} \tag{1}$$

The right hand side of equation 1 gives a rate difference divided by the congestion point's outgoing rate C giving a dimensionless number. In order for the left hand side to be dimensionless, $q(t)$ must measure time so that when divided by dt units cancel. $q(t)$ depends on C by $q(t) = \frac{L}{C}$ where L is the back log in bytes.

We can better motivate equation 1 as follows. L_0 is the initial backlog (bytes) at the congestion point at time $t = 0$. In practice time is measured in discrete time units typically μs. So assume $\Delta t = t_{n+1} - t_n = 1\mu$s:

$$\begin{aligned} q(t_0) &= \frac{L_0}{C}\mu\text{s} \\ q(t_1) &= q(t_0) + \frac{[y(t_1) - C]\Delta t}{C}\mu\text{s} \end{aligned} \tag{2}$$

Equation 2 gives the time in μs it'd take to drain all bytes in the congestion point after each tick in time. $y(t_1)\Delta t$ gives the number of bytes delivered by the M hosts during time tick t_1. During this same time, the congestion point processes $C\Delta t$ bytes. This difference when divided by C and added to the previous $q(t_0)$ gives the queue delay after t_1 concludes.

$\frac{dq(t)}{dt}$ can be approximated by taking the difference between two successive points in time divided by Δt.

$$\frac{\Delta q}{\Delta t} = \frac{q(t_{n+1}) - q(t_n)}{t_{n+1} - t_n} \tag{3}$$

The term $q(t_n)$ cancels due to recurrence. This simplifies to:

$$\begin{aligned} \frac{\Delta q}{\Delta t} &= \frac{[y(t_1) - C]\Delta t}{C} \cdot \frac{1}{\Delta t} \\ &= \frac{y(t_1) - C}{C} \end{aligned} \tag{4}$$

which agrees with equation 1, and is dimensionless.

Timely concludes section 4.2 by claiming the change in RTT between time ticks is just:

$$\frac{dRTT}{dt} = \frac{dq(t)}{dt} \tag{5}$$

This is the main claim. Timely references [3] noting:

> it is not possible to control the queue size when it is shorter in time than the control loop delay. This is the case in datacenters where the control loop delay of a 64 KB message over a 10 Gbps link is at least 51 µs, and possibly significantly higher due to competing traffic, while one packet of queuing delay lasts 1 µs. The most any algorithm can do in these circumstances is to control the distribution of the queue occupancy. Even if controlling the queue size were possible, choosing a threshold for a datacenter network in which multiple queues can be a bottleneck is a notoriously hard tuning problem.

For this reason Timely falls back to controlling the change in RTT. Note Timely does **not** depend on the packet's size. This issue is discussed further in section 5.

4 Timely Model

Timely runs the following event loop for each packet sent:

1. Fix initial sending rate at R, usually line rate
2. Send data at rate R
3. Measure the RTT r for data just sent
4. Compute new rate $R = \text{timely}(R, r)$
5. Goto 2

Congestion is managed through a time series of RTT values r_i by updating R. It reduces R when RTTs increase and vice versa.

The Timely model computes the new R by cases depending on how the last RTT r_i compares to a

model range $[D_{min}, D_{max}]$. When the $r_i < D_{min}$ the new rate R is additively increased. When $r_i > D_{max}$ the new rate R is multiplicatively decreased. And when $D_{min} \leq r_i \leq D_{max}$ a more complicated computation used.

Timely uses seven tunable values. We next describe each factor or case, then conclude with a succinct Timely algorithm.

4.1 EWMA α Factor

$\alpha \in (0, 1]$ is the weighting constant for an EWMA (Exponentially Weight Moving Average) of RTTs; see [8, p542] algorithm 1. Its purpose is to smooth RTT jitter by overweighting current RTT values. a_{n+1} denotes EWMA at step $n+1$ based on the last a_n where n indexes the packets sent. k_0 is the initial EWMA value typically zero. Δr represents the change in RTT between two successive steps: $\Delta r = r_{n+1} - r_i$:

$$\begin{aligned} a_0 &= k_0 \\ a_{n+1} &= \alpha \cdot \Delta r + (1 - \alpha) \cdot a_n \end{aligned} \tag{6}$$

Here's an example EWMA table. Note a_n changes less drastically compared to the actual RTT difference:

Iteration	**RTT μs**	Δr	a_n
0	351	0	0
1	50	-301	-144.47
2	352	302	69.83
3	83	-269	-92.81
4	86	3	-46.82

Table 1: EWMAs for $\alpha = 0.48, k_0 = 0$

4.2 Rate Decrease β Factor

The $\beta \in (0, 1]$ factor is used to decrease the new rate multiplicatively when the last RTT $r_{n+1} > D_{max}$. In equation 6 R_n is the last rate, R_{n+1} is the new rate. r_{n+1} is the RTT for the packet sent at step n:

$$R_{n+1} = R_n \left(1 - \beta \left(1 - \frac{D_{max}}{r_{n+1}}\right)\right) \tag{7}$$

When r_{n+1} exceeds D_{max} the term $1 - \frac{D_{max}}{r_{n+1}}$ gives a value in $(0, 1)$. This term is decreased by β through multiplication. In this way, the worse the RTT is, the smaller R_{n+1} becomes through the two $1 - x$ sub-expressions. And vice-versa: if RTTs are barely above D_{max}, the new R_{n+1} is decreased less.

4.3 Rate Increase δ Factor

This is the simplest case. When $r_{n+1} < D_{min}$ the new rate is simply:

$$R_{n+1} = R_n + \delta \tag{8}$$

4.4 Timely Rate Middle Case

When the last RTT $r_{n+1} \in [D_{min}, D_{max}]$ a more involved computation is performed. In this analysis we apply the patch developed in [4, p8] section 4.3 algorithm 2. This computation introduces a gradient, weight, and error sub-terms.

First, using the EWMA RTT value a_{n+1}, the gradient g_{i+1} is computed as:

$$g_{n+1} = \frac{a_{n+1}}{D_{minRTT}} \tag{9}$$

D_{minRTT} is a parameter which specifies the smallest RTT that should ever be considered for Timely rate updates.

Second, the weight w_{n+1} is computed through a piece-wise function:

$$w_{n+1} = \begin{cases} 0, & g_{n+1} \leq \frac{1}{4} \\ 2g_{n+1} + \frac{1}{2}, & -\frac{1}{4} < g_{n+1} < \frac{1}{4} \\ 1, & g_{n+1} \geq \frac{1}{4} \end{cases} \tag{10}$$

Next, the error term e_{n+1} is computed as:

$$e_{n+1} = \frac{r_{n+1} - RTT_{ref}}{RTT_{ref}} \tag{11}$$

[4] does not describe RTT_{ref} w. For purposes here regard it as equal to D_{minRTT}.

Finally, the new rate is computed from these interim calculations:

$$R_{n+1} = \delta(1 - w_{n+1}) + R_n(1 - \beta \cdot w_{n+1} \cdot e_{n+1}) \tag{12}$$

4.5 Timely Model Synopsis

Variable	Description
n	Indexes packets sent (zero based)
R_{n+1}	Rate (bytes/sec) for packet $n+1$
r_{n+1}	RTT for packet n (μs)
Δr	$r_{n+1} - r_n$ (μs)
a_{n+1}	EWMA for packet $n+1$ (μs)
a_0	Initial EWMA typically 0 (μs)
g_{n+1}	Gradient for packet $n+1$
w_{n+1}	Weight for packet $n+1$
e_{n+1}	Error for packet $n+1$
α	EWMA weight in $(0, 1]$
β	Rate multiplicative factor $(0, 1]$
δ	Rate additive factor (bytes/sec)
D_{min}	Timely model min RTT (μs) cutoff
D_{max}	Timely model max RTT (μs) cutoff
D_{minRTT}	Smallest measurable RTT (μs)
RTT_{ref}	Alias for D_{minRTT} (μs)
Z	NIC Bandwidth (bytes/sec)

Table 2: Timely Parameters. Items without explicit units are dimensionless

Combining the three cases, the Timely rate calculation event loop runs as follows:

1. Initialize $\alpha, \beta, \delta, D_{min}, D_{max}, D_{minRTT}$
2. Set $a_0 = r_0 = 0, R_0 = Z, n = 0$
3. Send packet n at rate R_n
4. Let r_{n+1} be the RTT for packet n
5. Let $\Delta r = r_{n+1} - r_n$
6. Let $a_{n+1} = \alpha \cdot \Delta r + (1 - \alpha) \cdot a_n$
7. If $r_{n+1} > D_{max}$: compute R_{n+1} from equation 6
8. If $r_{n+1} < D_{min}$: compute R_{n+1} from equation 7
9. If $r_{n+1} \in [D_{min}, D_{max}]$
   (a) Compute gradient g_{n+1} from equation 8
   (b) Compute weight w_{n+1} from equation 9 with g_{n+1}
   (c) Compute error e_{n+1} from equation 10
   (d) Compute R_{n+1} from equation 11
10. If $r_{n+1} > Z$ then $r_{n+1} = Z$
11. $n = n + 1$
12. goto 3

5 Carousel Motivation

Timely calculates the putative send rate for packet $n+1$ based on the RTT for last packet. However, this doesn't consummate an actual rate because it's only a calculation.

Consider three consecutive transmitted packets holding $[64, 1, 16]$ KB. Based on their RTTs, further suppose Timely calculated their respective rates as $[5, 1, 10]$ MBs (million of bytes per second). Assume the NIC bandwidth is 10Gbps. Now, the NIC can only serialize data at its bandwidth rate; there are no other options. So how might this data have been sent complying with Timely?

This is the purpose of Carousel [9]. Each outgoing packet in timestamped for egress then enqueued onto a *timing wheel*. The wheel behaves like a `std::priority_queue` where packets are ordered by release time. Carousel waits until the computer's clock equals an egress timestamp then releases ready packets to the NIC's transmission queue. This is called *traffic shaping*. See figure 1.

`eRPC` gives a favorable view writing "we implement Carousel's rate limiter, which is designed to efficiently handle a large number of sessions. Carousel's design works well for us" [11, p8].

Carousel provides per core wheels with deferred completion. Deferral bounds how much traffic hits

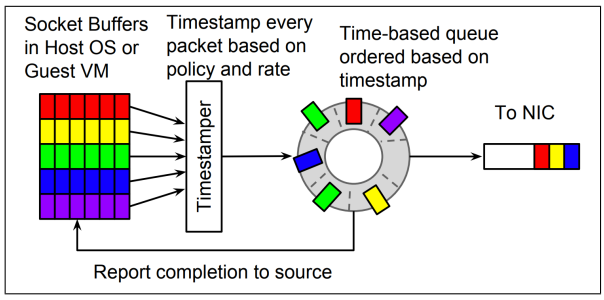

Figure 1: Carousel Design

the NIC without packet drops [9, p6] section 4. Its design comes in four parts discussed next.

First packets are timestamped. Second, packets are placed in a timing wheel. Third, the timing wheel must release ready packets sending them to the NIC. Finally, there must be a feedback mechanism to prevent enqueuing too much data, which is the purpose of deferral.

5.1 Carousel Timestamps

Consider again three consecutive packets holding $[64, 1, 16]$ KB with Timely calculated rates of $[5, 1, 10]$ MBs at the same 10Gbps line rate. If the NIC *could serialize* the first packet onto the wire at 5MBs, it'd take $(64 \cdot 1024)/(5 \cdot 5000000)$ or about 0.013 seconds. Now, provided the next packet is sent at least 0.013s after the completion of this first 64KB, the Timely rate of 5MBs is achieved for the first packet. Table 3 repeats this calculation for the other two packets.

Of course, the NIC only required about 52μs at 10Gbp to serialize this 64KB meaning the NIC was not busy for the last .01295s. There's a 64KB burst at line rate for the first 52μs then no work. Consequently, Carousel can only enforce average transmission rates. The NIC's bandwidth is fixed. So if Timely calculates a rate over this limit, it must be forced down to line rate.

There are two remaining conceptual problems. Note that releasing packets to the NIC ultimately comes down to a comparison between some packet's release time, and the computer's timestamp for now. If the release time is equal to or less than now, the packet must be sent otherwise held. This means the relative delay period alone is ambiguous.

Pkt#	Size KB	Rate MBs	Delay (sec)
0	64	5	0.013
1	1	1	0.001024
2	16	10	0.0016384

Table 3: Relative delay by packet at its target rate

The second problem involves multiple senders. Trivially each sender only knows about its packets, their size, and their delay periods. If the sender's share a timing queue, a software component must merge relative delays making an explicit absolute time ordering. Sharing is not imposed by Carousel. Sharing typically arises when the application wants to impose constraints on an aggregate of packets.

It's important to recall Timely's rate calculation remains substantially independent of Carousel. RTT is the only external signal that informs transmission rates for the next packet. By construction RTTs only arise in the context of one sender when it sends its next packet based on the RTT for the last packet. The one exception is when a Carousel policy appreciably changes a packet's release time which will indirectly impact the next RTT for some sender.

To address these concerns and fully resolve timestamps, we must first engage the issue of policy.

5.2 Carousel Polices and Timestamps

Time stamping packets works in conjunction with a policy to constrain when packets are sent. For example, an aggregate rate policy bounds how many bytes hit the NIC per unit time from all senders on the same CPU core. Pacing policies shape how many packets are transmitted per unit time.

In the original paper, Carousel contemplates multiple policies and wheels as data flows from a virtual OS into the bare metal OS then onwards into the TCP stack. We do not require those complexities. Kernel bypass with UDP short circuits the path from packet generation to packet transmission. Low latency high bandwidth RPCs usually runs on bare metal anyway.

We divide what work remains into four major sub-cases. Case one and three are analyzed to complete time stamping strategy. The other cases are presently considered impractical for `eRPC` goals. An analysis of `eRPC` source code shows it uses one Carousel object per RPC. There is no sharing. But, if required, the other solutions will involve typical multi-threaded synchronization combined with the timestamp consolidation explained in case three.

Refer to section 8 for important implementation notes on time tracking.

Case	Share Wheel	Enforce Policy
1	No	No
2	Yes	No
3	No	Yes
4	Yes	Yes

Table 4: Timestamp sub-cases

5.2.1 Case 1: No Share/Policy

This is the simplest case connecting one data producer with one Carousel object. It suffices for senders to enqueue packets with a relative delay period like table 3. The timing wheel can simply add the current time to the delay to create an absolute release ordering.

5.2.2 Case 3: No Share Aggregate Policy

Like case one there is one data producer paired with one Carousel object. Packets should be enqueued the same way. However, the resulting timestamps are subject to change when an aggregate send rate policy is enforced.

For example, suppose the aggregate policy enforces an upper transmission rate $M = 50$ KB/sec. M should be smaller than the NIC's bandwidth. Suppose a sender wants to transmit packets of size $[30, 40, 1, 16]$ KB with Timely rates of $[5, 1, 7, 10]$ MBs respectively. In this scenario, the timestamps for all but the first packet would have to delayed beyond what the send originally requested.

Once the 30KB packet is delivered to the NIC, the earliest the second packet (40KB) could be sent is after .0061440s delay leaving more than 99% of the first second free. However, M rate bound prevents sending the next 40KB at time .0061440 since $30 + 40 >$M.

And this is the salient difference compared to case one. This case requires the wheel to have efficient access to each packet whereas case one only needs efficient access to the packet with the smallest release time.

6 Delay Overview

When a data originator sends a network packet, the elapsed time until reception at the designated endpoint depends on several additive factors.

First, the sender has to make the packet. This can involve DNS, ARP, or other lookups to find the destination's MAC, IP address, and port. The sender must deliver the packet to its NIC through the operating system, `io_uring`, or a kernel by-pass library. This work competes with other processes on the same host machine. Denote by O the elapsed time here.

Second, and once data is in the NIC, the packet's ones and zeros are serialized or converted into electrical signals at the NIC's bandwidth rate B. This requires time $S = \frac{b}{B}$. For example, it requires 52μs to serialize 64Kb at 10 Gbps (10 Giga bits/second) or .8ns per byte.

Third, the electrical signals travel at the medium's (*ex.* fiber, cable, air) propagation rate r usually close to the speed of light. This delay is $P = \frac{d}{r}$ where d is the distance travelled. At this stage the packet is enroute to its destination.

Next, packets usually move through one or more intervening switches (OSI layer 2) or routers (OSI layer 3) before arriving at the final destination. Each hop requires overhead to deserialize, queue, process, and re-serialize each packet to the next hop. Routers must reconstruct part of the packet. It's slower than a switch. Designate by N_i the time delay per hop incurred including the propagation delay to forward the packet.

Finally, the end data receiver must deserialize the

electrical signals (at the receiver's NIC bandwidth), move the packet through the operating system or bypass library delivering it to application code for an additional contribution of R.

The total delay is the sum: $O + S + P + \sum_i N_i + R$ sometimes called a one-way RTT (round trip time).

7 RTT

Timely relies on accurate RTT signals. To be round trip, the originating sender requires an acknowledgement signal from the designated receiver.

Definition 7.0.1 (RTT)**.** *Suppose application code transmits b bytes in one packet at its NIC line rate B at time t_0 and receives an acknowledgment at time t_1. The RTT, which has units of time, for this packet is defined as $t_1 - t_0 - \frac{b}{B}$.*

Using the notation in section 6, this definition omits the overhead O and serialization time S at the sender frot time t_0. When RTTs are of the same order as S, or if O is high, RTT signal fidelity is corrupted. The timestamp t_0 must be taken when the last byte of packet is on the wire at the sender. However, all other time delays $P + \sum_i N_i + R$ are baked into t_1.

Timely does not engage the number of congestion points. In any commercial setting there's multiple hops. Senders and receivers run multiple processes all competing for hardware resources. Timely treats all those pieces as one virtual congestion point.

8 RTT, Timestamp, and Time

There are two viable ways to extract RTTs. Do not use software `clock_gettime()` or equivalents. They are not precise or repeatable enough. Use hardware.

8.1 Intel `rdtsc`

`eRPC` uses Intel's `rdtsc` processor instruction. Mention power-savings complications

8.2 NIC Timestamps

Discuss extracting timestamps for Mellanox via DPDK.

References

[1] DPDK. `https://www.dpdk.org`. [Online; accessed 01-Feb-2023].

[2] ACM Transactions on Computer Systems. *Full-Stack Architecting to Achieve a Billion-Requests-Per-Second Throughput on a Single Key-Value Store Server Platform* (Seattle, WA USA, Apr. 2016), vol. 34.

[3] Computer Communication Review. *Stability and fairness of explicit congestion control with small buffers* (Irvine, CA USA, 2008).

[4] CoNEXT. *ECN or Delay: Lessons Learnt from Analysis of DCQCN and TIMELY* (Irvine, CA USA, Dec. 2016).

[5] Kalia, A., Kaminsky, M., and Andersen, D. G. FaSST: Fast, scalable and simple distributed transactions with Two-Sided (RDMA) datagram RPCs. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)* (Savannah, GA, Nov. 2016), USENIX Association, pp. 185–201.

[6] NSDI. *FaRM: Fast Remote Memory* (Seattle, WA USA, Apr. 2014).

[7] NSDI. *MICA: A Holistic Approach to Fast In-Memory Key-Value Storage* (Seattle, WA USA, Apr. 2014).

[8] SIGCOMM. *TIMELY: RTT-based Congestion Control for the Datacenter* (Boston, MA USA, Aug. 2015).

[9] SIGCOMM. *Carousel: Scalable Traffic Shaping at End Hosts* (Los Angles, MA USA, Aug. 2017).

[10] SOSP. *KV-Direct: High-Performance In-Memory Key-Value Store with Programmable NIC* (Shanghai, China, Oct. 2017).

[11] USENIX Association. *Datacenter RPCs can be General and Fast* (Monterey, CA USA, Feb. 2019).

[12] Zhang, J., Shi, J., Zhong, X., Wan, Z., Tian, Y., Pan, T., and Huang, T. Receiver-driven rdma congestion control by differentiating congestion types in datacenter networks. In *2021 IEEE 29th International Conference on Network Protocols (ICNP)* (2021), pp. 1–12.