PostMan: Rapidly Mitigating Bursty Traffic via On-demand Offloading of Packet Processing

Yipei Niu, Panpan Jin, Jian Guo, Yikai Xiao, Rong Shi, Fangming Liu*, *Senior Member, IEEE,* Chen Qian, *Senior Member, ACM and IEEE*, Yang Wang

Abstract—Unexpected bursty traffic brought by certain sudden events, such as news in the spotlight on a social network or discounted items on sale, can cause severe load imbalance in backend services. Migrating hot data—the standard approach to achieve load balance—meets a challenge when handling such unexpected load imbalance, because migrating data will slow down the server that is already under heavy pressure. This paper proposes PostMan, an alternative approach to rapidly mitigate load imbalance for services processing small requests. Motivated by the observation that processing large packets incurs far less CPU overhead than processing small ones, PostMan deploys a number of middleboxes called helpers to assemble small packets into large ones for the heavily-loaded server. This approach essentially offloads the overhead of packet processing from the heavily-loaded server to helpers. To minimize the overhead, PostMan activates helpers on demand, only when bursty traffic is detected. The heavily-loaded server determines when clients connect/disconnect to/from helpers based on the real-time load statistics. To tolerate helper failures, PostMan can migrate connections across helpers and can ensure packet ordering despite such migration. Driven by real-world workloads, our evaluation shows that, with the help of PostMan, a Memcached server can mitigate bursty traffic within hundreds of milliseconds, while migrating data takes tens of seconds and increases the latency during migration.

Index Terms—Bursty Traffic, Packet Offloading, Packet Batching, High-performance Network Stack.

1 Introduction

Modern distributed systems usually scale by partitioning data and assigning these partitions to different servers [1], [2], [3], [4], [5], [6], [7], [8], [9], [10]. In such an architecture, some servers may experience a higher load than others, creating a classic load imbalance problem [4], [11].

Many works have studied how to mitigate load imbalance by better data partitioning and placement strategies [1], [4], [5], [7], [8], [10], [11], which work well for long-term and stable load imbalance. For load imbalance caused by unexpected bursty traffic, however, these approaches meet an additional challenge: to adapt to such unexpected load imbalance, we need to adjust the data partitioning or placement strategies online by migrating hot data to less busy servers, but migrating hot data will inevitably slow down the server hosting hot data—this is the server we want to accelerate. This means to alleviate load imbalance, these approaches will first exacerbate load imbalance for a while, which is a risk that production systems are often unwilling to afford. For example, Amazon Dynamo runs data migration at the lowest priority and finds that during a busy shopping season, data migration can take almost a day to complete [11]. Unfortunately, unexpected bursty traffic is frequently reported in practice, for various reasons such as a sudden event drawing public attention on a social network [12] or a hot item on sale [13], [14], [15], [16].

This paper proposes PostMan[1], an alternative approach to mitigate load imbalance for services that are processing small packets, which usually incur a high overhead for packet processing. Typical examples of such services include key-value stores and metadata servers. For example, Facebook reported that in its caching layer, most key sizes are under 40 bytes and the median value size is 135 bytes [17], [18]; metadata servers, such as NameNode in HDFS [2], are usually processing packets with a size of tens to a few hundred bytes.

The key idea of PostMan is motivated by the observation that there is a significant gap between the overhead of processing small and large packets because the networking stack has to pay a constant overhead for each packet, such as interrupt handling and system call. For example, when processing 64B packets on 10Gb Ethernet, Linux can achieve a throughput of about 1Gbps with a CPU utilization of 800% (8 cores with 8K concurrent connections) [19]. On the contrary, when processing 8KB packets with a CPU utilization of 800% (8 cores with 8K concurrent connections), Linux can

• *This work was supported in part by the NSFC under Grant 61722206 and 61761136014 (and 392046569 of NSFC-DFG) and 61520106005, in part by National Key Research & Development (R&D) Plan under grant 2017YFB1001703, in part by the Fundamental Research Funds for the Central Universities under Grant 2017KFKJXX009 and 3004210116, in part by the National Program for Support of Top-notch Young Professionals in National Program for Special Support of Eminent Professionals, and in part by the NSF grant CNS-1566403. (Corresponding author: Fangming Liu)*

• *Y. Niu, P. Jin, J. Guo, Y. Xiao, and F. Liu are with the National Engineering Research Center for Big Data Technology and System, the Services Computing Technology and System Lab, Cluster and Grid Computing Lab in the School of Computer Science and Technology, Huazhong University of Science and Technology, 1037 Luoyu Road, Wuhan 430074, China. E-mail: {niuypei, panpanjin, guojian, fierralin, fmliu}@hust.edu.cn*
• *R. Shi and Y. Wang are with the Department of Computer Science and Engineering, Ohio State University. Email: {shi.268, wang.7564}@osu.edu*
• *C. Qian is with the Department of Computer Science and Engineering, University of California Santa Cruz. Email: cqian12@ucsc.edu*

1. The source code of PostMan is released on Gitee: https://gitee.com/opencloudnext/PostMan.

achieve a throughput of about 10Gbps [19]. Newer networking stacks, such as mTCP [19] and IX [20], have mitigated this problem, but first, the gap still exists, though smaller, and second, a wide deployment of a new networking stack requires a big effort, because the networking stack is a critical component of the whole system, which may affect all other components.

Motivated by this observation, PostMan incorporates several middleboxes called helpers, which batch small packets for the server experiencing bursty traffic (called "helpee" in the rest of the paper), so that the helpee can enjoy the low overhead of processing large packets. This approach essentially offloads the constant overhead associated with each small packet from the helpee to the helpers, so as to improve the throughput of helpees. Hence, the overloaded helpees can enjoy the performance gain despite the additional data transfer via helpers, especially when the load so high that the bandwidth of helpees is saturated.

This approach brings several benefits: first, PostMan does not require time-consuming data migration. Instead, it only requires the clients to re-connect to the helpers, which can be completed within hundreds of milliseconds as shown in our evaluation. Second, PostMan can incrementally deploy new networking stacks on helpers and allow other servers to still use traditional stacks. Third, PostMan can use multiple helpers to accelerate one helpee, which means its capacity is not limited by the power of a single machine. Finally, offloading batching opens up new opportunities for further optimization: we observe that packets to the same destination have significant redundancy in their packet headers (e.g., same destination IP and port). By removing such redundancy, PostMan can achieve a considerable reduction in bandwidth consumption at the helpee.

Of course, PostMan has its limitation under specific scenarios: if load imbalance lasts long, PostMan will be more expensive than data migration because incorporating helpers incurs extra server resource. Therefore, we expect PostMan and data migration to be complementary to mitigate load imbalance: for unexpected bursty traffic, we can activate PostMan to accelerate the heavily-loaded server first; if such burst continues to happen regularly, we can migrate data since the machine is less busy with the acceleration of PostMan; after data migration is completed, we can disable PostMan to minimize cost. As a result, the helpers would not be active for a long time. Moreover, since PostMan only targets the servers experiencing bursty traffic and can efficiently process packets, we can use a few helpers for a large cluster to further reduce cost.

The idea of batching small requests to improve performance is certainly not novel. The key novelty of PostMan lies in its observation that, *for the purpose of mitigating unexpected load imbalance, batching should be performed remotely and on demand*: remote batching allows PostMan to accelerate a heavily-loaded server with the help of resource from other servers; on-demand batching allows PostMan to minimize the overhead by only helping those servers experiencing bursty traffic. To realize these ideas, PostMan includes four main features:

- We provide an efficient implementation of the helpers. By utilizing state-of-the-art techniques like DPDK and efficiently parallelizing work among multiple cores, a single helper node can assemble and disassemble around 9.6 million small packets per second. By removing redundancy in headers, PostMan can reduce packet header size from 46 bytes to 7 bytes: for 64-byte packets, this means about 50% higher bandwidth utilization at the helpee.
- To ensure packet ordering despite migrating connections across helpee and helpers and despite helper failures, PostMan keeps helpers *stateless* by maintaining sufficient information at the clients and servers to detect out-of-order packets and re-transmit packets when necessary. While we find many applications already implement similar functionalities, we provide a library to those which do not.
- To minimize the overhead brought by helpers, the overloaded server determines when clients connect/disconnect to/from helper nodes based on its real-time load statistics. Meanwhile, all the available helper nodes are automatically discovered and registered by Consul, enabling clients to select the idlest one from the available helpers to achieve load balancing.
- We present an adaptive batching mechanism to decide how many packets to assemble. It can increase the batch size for higher throughput under heavy traffic and decrease the batch size for lower latency under light traffic.

Our evaluation on Memcached and Paxos shows that, with the help of PostMan, the service can mitigate bursty traffic within hundreds of milliseconds, while migrating data can take tens of seconds. Further investigation shows that this is because PostMan can improve the goodput of Memcached and Paxos by $3.3\times$ and $2.8\times$, respectively.

This work significantly extends the previous work [21]. We re-design the mechanism of enabling/disabling PostMan, where the overloaded servers determine when clients connect/disconnect to/from helper nodes based on load statistics. Furthermore, we add a new feature for reconnecting and load balancing, which automatically discovers and registers all the active helper nodes. The servers send the latest list of active helper nodes to clients, when the clients require to re-connect to helpers or migrate connections. Then the clients inquire about the load statistics of the active helper nodes and adopt the roulette wheel selection algorithm to choose a helper to re-connect. In the previous work, the available helper nodes are hard-coded and a greedy load balancing strategy is employed. These extensions improve the flexibility of PostMan, achieving on-demand packet offloading and efficient utilization of helpers, which further minimizes the cost of helpers. Finally, experiments are added to evaluate the effectiveness of the extensions. Meanwhile, when evaluating the effectiveness of PostMan compared to data migration, we re-conduct a series of experiments with real-world traces reported by Facebook.

The remainder of this paper is organized as follows: Sec. 2 reviews the related work. We introduce the overall architecture of PostMan in Sec. 3. In Sec. 4, we demonstrate how PostMan batches small packets. We discuss the detailed implementation of PostMan in Sec. 5. In Sec. 6, we present the results of evaluating PostMan. Finally, Sec. 7 concludes the paper.

2 Related work

Data migration. Load balancing is a classic topic of distributed systems. Most existing systems use an adaptive approach to achieve load balancing: they can monitor the load of each machine and place new data on less busy machines [1], [4], [5], [7], [8], [10], [11], [22]. To mitigate load imbalance, these approaches have to split, merge, and migrate existing data partitions, which works well for long-term and stable load imbalance. Despite the support from such mechanisms, how to mitigate load imbalance caused by unexpected bursty traffic is still a challenge: since migrating data online will put more pressure on the machine that is already heavily loaded, the administrator is facing a painful trade-off between the short-term loss and the long-term gain of migrating hot data. PostMan can rapidly mitigate load imbalance without migrating hot data online. Moreover, PostMan relieves overload of helpees, offering sufficient time and resource for data migration.

An alternative solution is to cache hot data locally to reduce remote access to backend servers. However, data of the services (e.g., NameNode in HDFS and Memcached) that PostMan accelerates is frequently updated, so caching the data locally may incur the significant overhead of maintaining data consistency for large-scale clients.

Batching small packets. A classic method to improve the performance of processing small packets is to batch them to amortize the constant overhead across multiple packets. For example, TCP has the Nagle's algorithm to batch small packets. Modern NICs often use Generic Receive Offload (GRO) to re-segment the received packets. However, the power of such per-connection batching mechanisms is limited by the number of outstanding packets per client. In many cases, a client may have to wait for replies before it can issue new ones, and thus the number of packets that can be batched is limited.

Comet [23] batches the received data before batched stream processing at the server side. KV-direct [24] first batches multiple KV operations at the client side to increase bandwidth utilization, and then at the server side, it batches memory accesses by clustering computation together.

A few systems incorporate several nodes to batch packets for other nodes. For example, Facebook has built mcrouter to batch packets for its Memcached service [18]. NetAgg aggregates traffic along network traffics for applications following a partition/aggregation pattern [25]. MPI has a collective I/O mode to batch I/Os from multiple processes before writing them to disks [26].

Compared to these systems, PostMan uses batching for a different goal—mitigating unexpected load imbalance. To achieve this goal, PostMan offloads the overhead of batching from the heavily-loaded server to others and only performs such offloading when a server is under heavy pressure. These techniques allow PostMan to use extra resources to accelerate a server experiencing bursty traffic and minimize the overhead when there is no such bursty traffic.

Efficient network stack. There is a continuous effort to develop more efficient network stacks for high-speed networks: mTCP [19] moves the TCP stack to the user space to reduce system call overhead and further improves performance by batching I/Os; DPDK [27] asks a network card to transfer data to the user space directly and applies a series of optimizations like CPU affinity, huge page, and polling to get close to bare-metal speed; IX [28] and Arrakis [29] design new operating systems to separate data transfer and access control to achieve both high speed and good protection; Netmap [30] improves networking performance by reducing memory allocation, system call, and data copy overhead.

Although these works have significantly improved the network performance, the performance gap between small and large packets persists (Table 1). Taking IX [28] as an example, it can achieve almost 10Gbps bandwidth even with 64-byte packets, which is significantly better than Linux. However, first, it needs to consume a considerable amount of CPU resources (see Section 6.2); second, there is still a gap between goodput (bandwidth used for payload) and throughput, because packet headers consume a large portion of bandwidth. Such per-packet overhead exists in RDMA systems as well [31].

Many works [32], [33], [34], [35], [36], [37], [38], [39], [40], [41] exploit the high-performance hardware to improve networking performance. These devices (e.g., smartNIC, RDMA, and NVMe) are not widely available yet. Furthermore, due to limited programmability and necessary adaptation of software stacks, it is hard to deploy the devices quickly. PostMan is a software solution, which can be quickly deployed and rapidly mitigate load imbalance with a minimum of cost.

Furthermore, the deployment of new networking stacks is usually a slow procedure, because networking service is critical and production systems are unwilling to pay any risk. On the other hand, PostMan allows administrators to incrementally deploy such new techniques on a few servers to accelerate a large number of legacy servers.

	64 bytes	64KB
10Gb Linux	2.4Gpbs	9.1Gbps
10Gb IX [28]	5.0Gbps	9Gbps

TABLE 1: Goodput of processing big and small packets. Goodput excludes bandwidth used for headers.

Others. The architecture of PostMan is similar to existing proxies (e.g., NGINX [42] and mcrouter [18]), which are also deployed between clients and servers. The key difference is that PostMan dynamically enables and disables helpers according to the load of servers.

The design of PostMan may seem to be similar to that of the split TCP approach [43], [44], [45], which also deploys helper nodes in the network. However, their goals and internal mechanisms are different: split TCP is designed to reduce latency in a network with large round-trip delays, by letting helper nodes send *ack*s to the sender directly; PostMan, on the other hand, is designed to improve the throughput of transferring small packets by letting helper nodes batch small packets. For the purpose of tolerating helper failures, PostMan's helpers actually delay sending

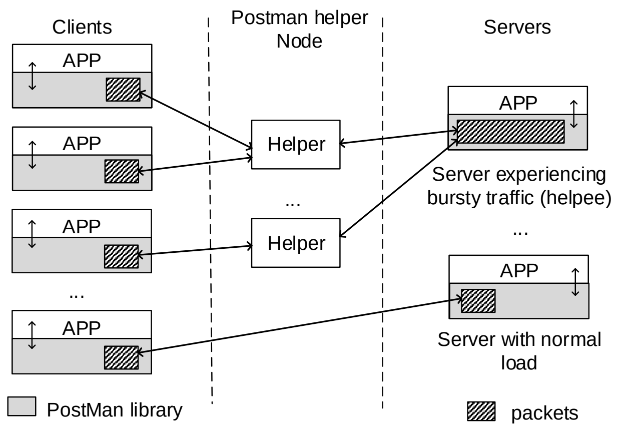


Fig. 1: Overview of PostMan.

*ack*s to the sender, until the helpers receive *ack*s from the helpee (see Section 4.2).

3 Overview of PostMan

In this paper, we propose PostMan, a distributed service to offload the overhead of packet processing from a heavily-loaded server (called helpee in the rest of this paper). Motivated by the observation that processing large packets incurs far less overhead than processing small packets, PostMan deploys a number of helper nodes in the network to assemble small packets for the helpee. By doing so, PostMan essentially offloads the constant overhead associated with each packet from a helpee to the helpers. With the help of PostMan, a helpee node only needs to process large packets.

Figure 1 shows the organization of PostMan. The core of PostMan consists of: 1) a number of helper nodes that assemble small packets for the helpees, and 2) a PostMan library that provides the applications with the functionalities of packet re-direction, assembly, and disassembly. Furthermore, PostMan library provides functions to detect out-of-order packets and re-transmit packets when necessary. These functions allow PostMan to achieve fault tolerance and load balance by migrating connections across helpers.

As shown in Figure 1, in a large scale distributed system, PostMan will only activate helpers to accelerate servers experiencing unexpectedly high load, which causes their latency to be higher than their service level agreement (SLA). For servers with normal load, their clients should communicate with the servers directly. Accelerating these normal servers with PostMan, though possible, is not cost-effective. Essentially PostMan *offloads* overhead instead of *reducing* overhead: in fact, PostMan increases overall overhead because it needs to perform additional work to assemble and disassemble packets. Therefore, PostMan tries to keep such overhead low by only helping nodes with trouble.

4 PostMan Design

PostMan is designed for the scenario that, suddenly, a large number of clients are sending small requests to a few servers (i.e., helpees). PostMan deploys helper nodes to assemble the clients' small packets to the helpee and disassemble the helpee's small packets to the clients so that the helpee only

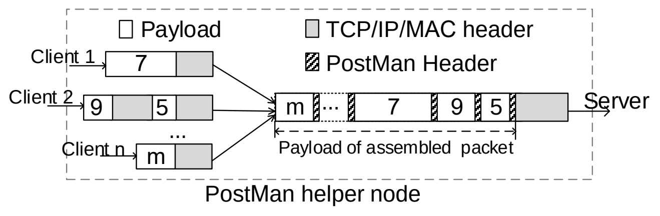


Fig. 2: PostMan assembles packets from different clients by using a short header for each payload.

needs to process large packets. To differentiate these two directions, we use "request" to refer to a packet from a client to a server and "reply" to refer to a packet from a server to a client.

In the rest of this section, we discuss how to assemble and disassemble packets efficiently at helper nodes, what APIs PostMan provides and how to apply them, and how to adaptively balance throughput and latency.

4.1 Assembling and disassembling packets

Format of assembled packets. For small packets, the size of their headers (at least 20 bytes for IP and TCP header respectively, 6 bytes for MAC header) is comparable to the size of their payloads, and that is one major reason why network throughput cannot reach bare-metal bandwidth, even with new techniques like DPDK. However, when considering packets to the same destination, their headers contain a significant amount of redundancy: packet assembly at the helper nodes can remove such redundancy and further improve throughput at the helpee. For example, since packets to be assembled have the same destination, PostMan only needs to maintain one copy of destination IP and port in the assembled packet. PostMan can shrink source IP as well for small to medium clusters by maintaining a mapping from IP to a shorter identification number at each node (e.g., 2 bytes for clusters with less than 64K machines). Moreover, since the connections from the clients to the helper and the connections from the helper to the helpee are separate, they perform congestion control independently, which means the helper can simply discard related information in the original packets.

As shown in Figure 2, a helper node can assemble packets from multiple connections, and when doing so, the helper discards their TCP/IP/MAC headers and only sends their payloads, together with one TCP/IP/MAC header for all payloads, so that the helpee does not need to pay the header overhead for each packet. However, to ensure that the helpee can correctly disassemble packets, the helper node must encapsulate necessary information for each payload, which is called a "PostMan header".

A PostMan header is a 3-tuple structure (Table 2): 1) an identification code to identify the packet type, 2) a length field to record the length of one payload, and 3) the source IP and port of the payload to locate the sender. A packet can have one of the following three types: 1) *request*, i.e., a packet sent by a client, 2) *reply*, i.e., a packet sent by a server, and 3) *connect*, i.e., a command to create a connection (see Section 4.2). As a result, compared to a TCP/IP/MAC header that

Type	Length (bytes)	Description
ID code	1	Message type
Len	2	Message length
Sender	2* + 2	Src IP/port

TABLE 2: The format of PostMan header. (*: for a cluster with less than 64K machines, the helper extracts the lower 16 bits from a source IP and then hashes them into a 2-byte identifier)

takes at least 46 bytes, a PostMan header only takes 7 bytes: this is a significant saving when processing small packets.

Workflow of assembling and disassembling packets. When assembling packets from the clients to the helpee, a helper node fetches all pending packets in its network stack, replaces their TCP/IP/MAC headers with corresponding PostMan headers, concatenates all of them, and adds its own TCP/IP/MAC header. By doing so, both the PostMan headers and the payloads of the original small packets become the payload of the assembled packet.

In the opposite direction, when a helpee sends replies to clients, it will first send the assembled reply to the helper node, which will disassemble the replies and dispatch them to the clients. The format of the assembled reply is similar to that in Figure 2.

Each helper node can create multiple connections to the helpee, so that one helpee can utilize multiple cores and threads to receive packets concurrently.

If one helper node is accelerating multiple overloaded servers, packets will be sent from clients to different destinations through helpers. For each helper node, the small packets, which are sent to the same destination (i.e., one of the overloaded servers), will be assembled into large packets. For packets sent to different destinations, PostMan does not assemble them at all. Hence, if the number of packets sent to each server is too small to assemble, PostMan fails to batch any packet. However, under such a scenario, the load is so low that PostMan has already been disabled.

4.2 PostMan library

A traditional networking library provides a number of APIs, such as *bind*, *connect*, *send*, and *recv* to the application. As shown in Figure 3, PostMan library provides a few additional ones: *pm_connect* allows a client to create a connection to a helper node; *compose*/*decompose* assemble/disassemble packets as described in Section 4.1; *get_info* allows the application to retrieve connection information. A developer should use these additional APIs together with traditional APIs to build the application. Next, we show how these functions work and how to modify an application to utilize these functions.

Establish connections. A server should bind to a port and wait for new connections, like using a traditional network library. Of course here a server may accept new connections from the helper nodes. A client can use the traditional network library when the latency is low and switch to PostMan when the latency is high by calling *pm_connect*. *pm_connect* will choose a helper (see Section 5.2) and connect to the helper. Then it sends a special “connect” packet to the helper node. This packet contains the destination IP and port of the helpee and the source IP and port of the client.

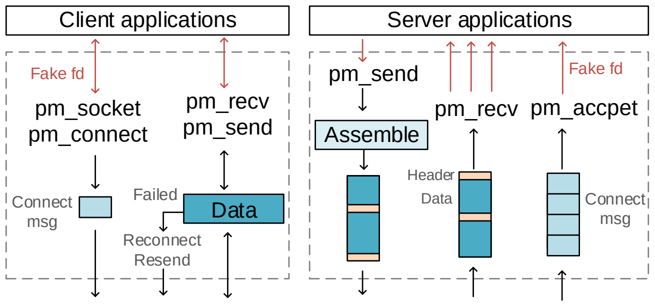


Fig. 3: Workflow of the PostMan library.

The helper node will connect to the corresponding helpee, if there is no connection yet, and forward the “connect” packet. At the same time, the helper creates a mapping from the client’s socket to the server’s socket. When the server application reads data from a helper connection, it should call *decompose*, which will identify the special “connect” packet and notify the server application that a new client tries to connect. Then the server application calls *pm_accept* to create a fake socket with a fake fd for the client connection and maps it to the helper connection using the source IP and port information. Finally, the server library will return a “success” packet to the client through the helper node, telling the application *pm_connect* succeeds.

Transfer data. When PostMan is activated, a client should send packets through the connection to the helper. The helper node assembles multiple small packets and sends the assembled packet to the server. When the server application reads a packet, it calls *decompose* to disassemble the packet into small packets. For each small packet, the library on the server side finds the corresponding fake fd according to the source IP and port information in the PostMan header. Then the small packets are delivered to the server application and processed as if they were from real sockets. In the opposite direction, when a server sends replies, for each helper connection, it should buffer multiple replies from the corresponding fake sockets and assemble them by calling *compose*. Then it can send the assembled reply to the connection to the helper. The helper node disassembles the replies and sends them to the corresponding clients based on the PostMan headers. The clients can read packets using a *recv* or *read* system call.

Ensure packet ordering. Applications often need to ensure packets are not lost, duplicated, or re-ordered. Since PostMan uses TCP connections between the clients and the helper nodes, and between the helper nodes and the helpees, these properties hold when there is no migration of connections. However, PostMan may migrate connections for several reasons: 1) If a client is connecting directly to a server and the server finds the throughput is high, it will notify the client of calling *pm_connect* to migrate its connection to a helper; 2) When a helper is heavily loaded, PostMan will instruct its clients to migrate to other helpers; 3) If a helper fails, its clients have to migrate to other helpers and recover the connections. As a result, PostMan needs additional mechanisms to ensure packet ordering despite such connection migration. For the first two cases, where migration is executed gracefully, a simple solution is to ask a client to wait for replies of all its pending requests before migrating its connection. For the helper failure case, however, this problem becomes challenging, because a client is uncertain about which packets are delivered.

In distributed systems, two approaches are widely used to achieve fault tolerance: one is to replicate the nodes that can be faulty, and the other is to re-direct requests to another node. Replication can fully hide faults from upper layers, at the cost of increased overhead. The re-direction approach has lower overhead, but it requires the faulty node to be *stateless*, i.e., it does not have any important state that will affect execution.

We use the re-direction approach because of its low overhead. To ensure packet ordering even if the system loses all information on the faulty helper node, PostMan needs to maintain sufficient information at the senders and receivers. Its basic idea is similar to that of TCP: a sender should buffer a packet until it gets acknowledged and re-send a packet if it does not get acknowledgment in a given amount of time; a receiver should check the sequence numbers in the packets to ensure they are in order. Unlike TCP that implements this mechanism in one connection, PostMan needs to implement this mechanism across different connections, because when a helper fails, the client needs to re-connect to a new helper.

On one hand, we observe that many applications have already implemented such a mechanism. The fundamental reason they choose to do so instead of relying on TCP is that they are designed to tolerate machine failures: in this case, a node needs to connect to other nodes and face the same problem as PostMan. For these systems, PostMan can utilize the application's mechanism directly.

On the other hand, for applications that do not have this mechanism, PostMan provides a general solution. To ensure packets will not be lost, PostMan library at senders[2] will buffer packets until it receives *acks* from receivers, as the TCP protocol does. Since the underlying layer maintains separate TCP connections between the senders and the helper nodes, and between the helper nodes and the receivers, the key to avoid data loss is to coordinate the underlying *ack* mechanisms: after receiving a packet from a sender, the helper node should not send the *ack* to the sender until it has got *ack* from the receiver. We modify the TCP implementation at the helper nodes to realize this mechanism. Since Postman targets small packets, delaying acks and sending them in burst should have little impact on congestion control.

When a helper fails, a sender may not receive *acks* for its outgoing packets, so it may decide to reconnect to another helper and retransmit those packets through the new helper. In this case, since the receiver may have already received these packets (*acks* may be lost due to helper failure), these packets may be duplicated or re-ordered. To prevent such abnormality, PostMan libraries at the sender and the receiver maintain additional information to detect duplicate or out-of-order packets.

To be specific, the library will keep track of how many bytes are already sent and received on each connection; for each buffered outgoing packet, the library will record its offset in the stream. As shown in Figure 4, when a helper node fails, the client library, which is the sender in this example, will connect to another helper node and sends a "reconnect" message to the server through the new helper node, which contains the number of sent and received bytes at the client side. When the server library receives this command, it will first stop receiving packets from the old connection and then respond with the number of sent and received bytes at the server side. By exchanging the number of sent and received bytes and comparing them to the offsets of buffered packets, both sides can determine which packets should be re-transferred.

2. Note that senders and receivers are different concepts compared to clients and servers: when a server is receiving packets from a client, it is the receiver; when a server is sending a reply to a client, it is the sender.

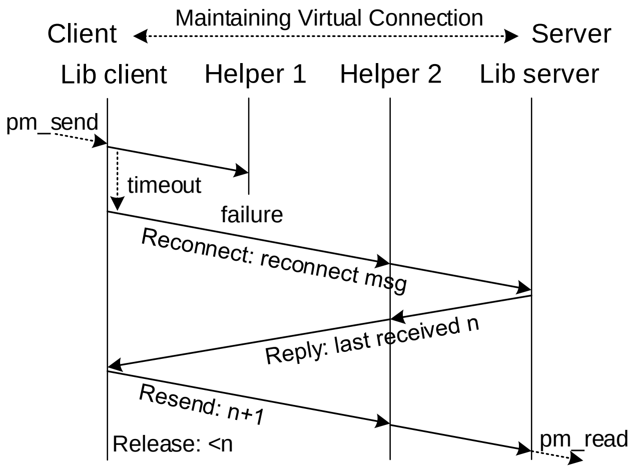


Fig. 4: Maintaining a virtual connection by redirecting and re-sending requests when a helper node fails.

Further optimization. So far we assume an application server needs to disassemble packets before processing them. However, this may not be necessary for some applications. A typical example is a server that needs to forward or broadcast packets (e.g., proxy server, leader replica in replication protocols, etc). For such servers, since they do not care about the content of payload, they can forward or broadcast the assembled packets directly, instead of disassembling them first. Note that when sending assembled packets, the application should not use PostMan, since these packets are large.

Using PostMan library. To apply PostMan to existing applications, the developer needs to modify its code: at the client side, the client should call *pm_connect* to switch to PostMan when it observes a high latency and switch back to traditional sockets when the latency drops back to normal; at the server side, the server should call *decompose* when it receives a packet from a helper (*get_info* can tell whether a connection is from a client or from a helper). The server assembles a number of replies by calling *compose* when it sends packets through helpers. If the application needs PostMan's help to ensure packet ordering, it should notify PostMan when a packet is sent or received, so that PostMan can buffer and release packets and update corresponding metadata.

It is possible to hide all the mechanisms mentioned above in the library and provide the applications with an illusion that they are using direct connections between

Algorithm 1 Adaptive batching algorithm

Input: the size (s) and waiting time (t) of last batch

```
1:  procedure Update $S$ and $T$
2:      $S_l \leftarrow 0.75S$
3:      $S_u \leftarrow 1.25S$
4:      if $(s < S_l \vee s > S_u) \wedge (s \geq 1500)$ then
5:          $S \leftarrow s$
6:      end if
7:      $T_l \leftarrow 0.5T$
8:      $T_u \leftarrow 1.5T$
9:      if $(t < T_l \vee t > T_u) \wedge (t \geq 10\mu s)$ then
10:         $T \leftarrow t$
11:     end if
12: end procedure
```

clients and servers. We have implemented a library to achieve such transparency. However, we find it can incur up to 50% overhead for additional operations like memory copy, synchronization, context switch, etc. Considering the main goal of this work is to improve the performance of the heavily-loaded server, we decide to give up transparency for better performance.

4.3 Adaptive batching

Batching can affect system latency in two opposite ways: on one hand, to assemble packets, a helper node must wait for a certain amount of time to accumulate enough small packets, which will increase the latency of the system. On the other hand, according to queuing theory, when the load is close to or higher than the system's capacity, queuing delay will become a dominant factor for latency. Since batching can improve a system's capacity, it can reduce queuing delay and thus can reduce latency.

PostMan partially avoids such trade-off by only activating helpers for heavily-loaded servers. In addition, PostMan incorporates an adaptive batching algorithm to balance latency and throughput when helpers are enabled. Like many systems using batching, PostMan defines a maximum batch interval (T) and a preferred batch size (S): if the helper has waited for T (condition 1) or if its assembled packet has reached size S (condition 2), the helper will send the assembled packet to the helpee. Then the question turns to how to set T and S: large T and S lead to unnecessary waiting when traffic load is light; small T and S reduce the chance of assembling packets when traffic load is heavy.

To address this problem, PostMan uses an adaptive batch size and interval to increase throughput under heavy loads and decrease latency under light loads, as shown in Algorithm 1. PostMan records the batch size (s) and waiting time (t) of the last batch: if s is significantly different from S or t is significantly different from t, PostMan updates S and T accordingly. Furthermore, it sets a lower bound of S and T to ensure efficiency. Note that although T is the maximum batch interval, the actual interval t can be much larger than T when the helper does not receive any packets for a long time; s can be much larger than S as well when the helper receives many packets at the same time.

This algorithm has a few parameters: we set the lower bound of S to be 1500 because that is the MTU size; the lower bound of T should be set according to the SLA requirement; we set other parameters based on empirical experiments.

5 Implementation

In this section, we present how to achieve efficiency and scalability for PostMan.

5.1 Efficient helper

Fast I/O and user-level stack. Each helper node needs to assemble requests from the clients, and disassemble the replies from the servers. To efficiently process the small packets on the helper nodes, we implement PostMan upon DPDK [27], which is a set of libraries and drivers for fast packet processing. DPDK minimizes the overhead of packet processing by transferring packets from NICs directly to user space programs and thus can achieve a throughput of hundreds of millions packets per second. Upon DPDK, we use mTCP [46] to handle TCP protocol and connections. DPDK provides a poll mode I/O model, which can transfer a batch of packets in one I/O operation. This I/O model not only avoids the overhead caused by frequent interrupts in per-packet based processing in Linux, but also naturally fits the assembling requirements of our helper nodes: a helper can simply add all the payload data from these packets to the assembling buffer, instead of performing the read operation several times.

In-stack processing. Since the assembling logic only involves simple operations for request/response headers, PostMan implements these operations in the network layer to accelerate the identification and pre-processing of the original packets. PostMan uses direct data exchange between the server and the client streams so as to avoid redundant memory copy and improve the performance of fragmented data operation. By implementing everything in the network stack, PostMan eliminates the interaction and context switching between the applications and the stack to further improve assembling efficiency. All necessary assembling and disassembling operations are queued in the stack, so that PostMan can perform them after finishing processing the incoming packet in the TCP protocol. Furthermore, PostMan only keeps the necessary procedures for receiving and sending packets in a TCP stream. Other operations, like the ICMP protocol, are abandoned, either because they have nothing to do with packet assembly, or they should be performed in the helpee's stack.

Independent per-core context. PostMan leverages per-core thread (affiliated to a hardware thread) and independent per-core context to avoid synchronization among different assembling threads. On each helper node, we enable the Receive Side Scaling (RSS) [47] function of NIC—this is widely supported by today's NICs—to hash flows into different physical queues in hardware, where each queue is assigned to a dedicated CPU thread. PostMan does not share any global information, hence the connection can be locally processed on each core. PostMan sets up at least one queue for each thread, such that the flows in each RSS group can be processed independently without exchanging any information between CPU cores. Consequently, PostMan can scale well on today's multi-core system. Note that RSS will reduce a helper's chance to batch packets, but since PostMan is enabled only when the server is under high load, there are still plenty of chances for a helper to batch packets as shown in our evaluation. For each thread, PostMan uses hugepages to store raw packet data, similar as many DPDK based applications, so as to reduce the number of TLB misses; PostMan uses hardware-based CRC instruction for flow hashing to accelerate the assembling process.

5.2 Load balancing

As presented in the previous section, PostMan is designed for hardware thread and RSS built-in NIC. Hence, PostMan scales well with the number of cores.

We use Consul [48] to automatically register and discover the helpers nodes. Consul is a distributed and highly available solution to automate network configurations, discover services, and enable secure connectivity across any cloud or runtime. Combining Consul with consul-template [49], a client can get the available helper nodes from the consul-template file when re-connecting. For helper nodes, their stateless nature, which allows connections to be migrated freely across helpers, significantly simplifies scaling and load balancing: when a client connects to helpers, it randomly selects one. Whenever some helper nodes are overloaded, they can simply disconnect some clients and those clients will automatically re-connect to other helper nodes. To achieve this, PostMan uses a simple yet effective load-balancing strategy: each helper monitors its own thoughtput and memory utilization. When a client establishes a connection, either for a new connection or for re-connecting, it inquires about the load statistics of all the available helpers (except the one just disconnected from); each helper will reply with its resource utilization, so that the client can choose a helper using the roulette wheel selection algorithm based on load; when a helper finds its resource utilization is too high, it disconnects existing connections, so that the corresponding clients can connect to other helpers.

PostMan has no inherent scalability bottleneck: since a client can connect to any helper, PostMan does not need a centralized master node to map clients to helpers.

5.3 When to enable and disable helpers?

PostMan provides a mechanism to enable and disable helpers on demand, but in practice, we still need to answer the policy question about when to enable and disable helpers. In principle, both clients and servers can make the decision and we observe the following trade-offs: A client can monitor its perceived latency to the server and make decisions accordingly: this approach brings minimal overhead to the server side, but since a client cannot gain the overall load statistics of the server, it may not be able to make the best decisions in certain cases. For example, if the clients perceive that the latency exceeds a certain threshold, the clients would disconnect from servers and re-connect to helper nodes. However, the increase of latency on the client side may be caused by various issues, not only load imbalance on the server side. The clients hence may make the false decision on enabling PostMan. On the other hand, a server certainly has more information to make a better decision, but to execute the decision, the server must pay the overhead of notifying corresponding clients and helpers, which could be problematic if the server is already under heavy load. Nevertheless, notifying clients to disconnect from the overloaded server is the first step to mitigating bursty traffic. Furthermore, since the clients cannot acquire the overall system statistics, only the servers have sufficient load information to make the decision. Hence, compared to the previous version, when to enable/disable PostMan is left to servers.

Specifically, when the server detects its throughput is nearly saturated, it will send packets consisting of the "re-connecting" message and the list of active helper nodes to the connected clients via UDP. After receiving the information, the clients will disconnect from the server and re-connect to helpers (how to choose a helper is demonstrated in Sec. 5.2). On the other hand, a server will disconnect its helpers if its throughput becomes low (e.g., when data migration is finished or the load decreases), and send the notifications to the clients via UDP as well. If the messages are lost, the clients will keep the current connections (connections to servers or connections to helpers). As a result, if the server still receives packets from clients/helpers. The server would send the messages again till the sources of the packets change. Sending UDP packets to notify the clients will inevitably incur certain overhead, especially when the bursty traffic is arriving. However, the overhead can be negligible even when the server is overloaded, which is verified in evaluation.

The destinations of packets do not affect the decision of enabling/disabling PostMan. For each backend server, if the throughput of the server exceeds a threshold, PostMan will be enabled and the server will connect to helper nodes. Otherwise, the server will disconnect from the helper nodes. PostMan will be disabled until there is no server connecting to the helper nodes.

6 Evaluation

The goal of PostMan is to quickly mitigate the bursty traffic directed to one or a few servers. To assess whether PostMan achieves this goal, we evaluate the performance of PostMan using various applications and workloads. In particular, our evaluation answers the following questions:

- How well can PostMan help a service reduce the load caused by bursty traffic?
- How is PostMan's capability affected by packet size?
- How much resource does PostMan need to achieve such benefit?
- How does the system perform when there are helper failures?

To answer the first question, we run benchmarks on Memcached and Paxos, and emulate the case of bursty traffic by drastically increasing the load during the middle of the experiment; we enable PostMan after such burst to measure 1) whether it can reduce the latency of the target service and 2) how long it takes to enable PostMan. We compare the results of PostMan to those of the data migration approach.

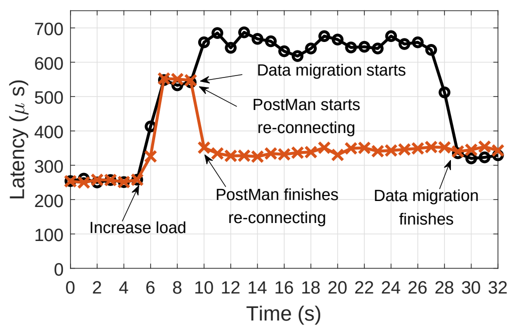


Fig. 5: Mitigating bursty traffic in Memcached (PostMan enables two helpers).

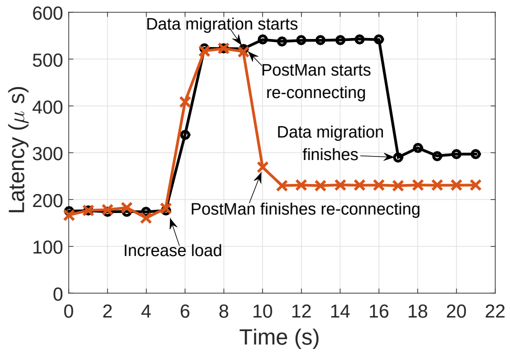


Fig. 6: Mitigating bursty traffic in Paxos (PostMan enables two helpers).

To answer the following three questions, we use a ping-pong microbenchmark to measure the performance of PostMan under different parameters.

Memcached. Memcached is a key-value based memory object caching system [50]. It is used widely in the data centers to cache data to speed up the lookups of frequently accessed data. As reported in [17], Memcached is often used to store small but hot data. We have modified 454 lines of code in Memcached 1.4.32 to apply PostMan. The benchmark generates GET/SET commands with a fixed key size (32 bytes) and different value sizes.

Paxos. Paxos is an asynchronous replication protocol to achieve consensus in a network of unreliable processors [51]. Paxos needs $2f+1$ replicas to tolerate f machine crashes and asynchronous events (e.g inaccurate timeout caused by network partitions). A number of systems, such as Megastore [52], Windows Azure [7] and Spanner [8], are using Paxos for fault tolerance and since they use many Paxos groups, one for each data partition, it is possible that a few of them experience bursty traffic.

In Paxos, one replica is elected as leader, and it needs to broadcast the received requests to other non-leader replicas. It is a typical example of applications that do not care about the contents of packets. Therefore, the leader can read the assembled packets from PostMan and broadcast the assembled packets directly. After the non-leader replicas receive the assembled packets, they will disassemble them. Such mechanism can avoid the redundant disassembling and assembling operations at the leader replica, which is the bottleneck in the system. To exploit such opportunities, we implement our own version of Paxos using PostMan and compare it to a vanilla version that reads individual packets from the clients, assembles them, and then broadcasts the assembled packets. For simplicity, we only implement the Paxos protocol in the failure-free case, which is enough to evaluate the performance benefit of PostMan.

Ping-pong benchmark. This benchmark [28] can test network performance with configurable packet sizes. To avoid the inaccuracy caused by TCP merging packets, this benchmark asks each client to perform a ping-pong like communication with the server: the client sends a packet to the server and then waits for the server to send the packet back. By doing so, since a client has only one outstanding packet, TCP has no chance to merge packets.

Experiment Setup. We run all experiments on CloudLab [53] with 15 machines. Each machine is equipped with an Intel Xeon E5-2660 v3 @ 2.60GHz CPU, with 10 physical cores and hyper-threading, and an Intel 82599ES 10-Gigabit NIC. These severs run Ubuntu 16.0.2 LTS with Linux 4.8.0 kernel, and use DPDK 16.07.2 for the helper nodes. For DPDK Poll Mode Driver, we set the batch size to 64, which is the maximum number of packets received/transmitted in each iteration. For Paxos and Ping-pong experiments, we use IX [28], a state-of-the-art network stack built upon DPDK, at the client side, so that we can stress-test the server with a limited number of client nodes. We also add 17 LoC to count the RX/TX bytes and packets in the data plane of IX, whose impact on the performance is negligible in our experiments. For Memcached experiments, since the Linux stack is sufficient to saturate the server, we do not switch to IX. By default, the application server of our benchmarks runs on 16 cores (8 real cores with hyperthreading). Note that in all experiments, when enabling PostMan, our reported goodput does not include the PostMan header and the TCP/IP/MAC header added by the helper nodes, which allows a fair comparison with the goodput without PostMan. When using PostMan, a client enables helpers if its observed 99 percentile latency (p99) is higher than 500 μs [20].

6.1 Effectiveness of PostMan

To measure the effectiveness of PostMan, we emulate the case of bursty traffic on both Memcached and Paxos.

Memcached. We measure the p99 latency of Memcached using GET requests with 32B keys and 64B values. As shown in Figure 5, we first use a light load, which incurs a latency of $200\mu s$, till the 5th time slot. Then we increase the load drastically, which increases the latency to more than $500\mu s$. At about the 9th time slot, we enable PostMan, which asks clients to re-connect to helpers. Such re-connection involves 660 client connections and finishes within $550ms$. Afterwards, the latency is reduced to around $300\mu s$.

As a comparison, we emulate the data migration by assuming 50% of the clients are accessing of the 10% of the keys (i.e., 100K keys in this experiment) in the Memcached server. We introduce memcache-mover [54], a simple tool to help copying the contents of a Memcached server into a new one. Such a tool has two modes: the destructive mode deletes the keys from the original Memcached after copying;

the non-destructive mode lists the keys before copying and keeps the keys in the original Memcached. Since we configure memcache-mover with non-destructive mode to keep the Memcached processing requests while migrating keys in this experiment. Therefore, we start a memcache-mover thread in the Memcached server to copy the key-values to another server at the 9th time slot. Such a thread needs to access the internal data structure of Memcached, which may incur additional CPU overhead and contend with client's requests. After data copy is complete, we remove half of the clients since they should be re-directed to another server. As one can see, the data migration takes about 13 seconds and during this procedure, the latency of the service further increases, because the migration traffic and the client's traffic compete for resource. One can of course limit the rate of migrating data to reduce such interference, but that will further increase the length of data migration.

Paxos. We run a similar set of experiments on Paxos. We measure the p99 latency of Paxos using a workload with 64B requests (Paxos does not execute the requests, so the content of the requests does not matter). As shown in Figure 6, we first use a light load, which incurs a latency of $200\mu s$, till the 5th time slot. Then we increase the load, which increases the latency to about $500\mu s$. At about the 9th time slot, we enable PostMan, which asks clients to re-connect to helpers. Such re-connection involves 960 client connections and finishes within $750ms$. Afterwards, the latency is reduced to around $220\mu s$.

Similarly, we emulate the data migration approach by assuming 50% of the clients are accessing 10% of the data. Therefore, we start a thread in the non-leader server to copy the data to another server at the 9th time slot. Since the leader has the highest overhead in Paxos, copying data from a non-leader server should incur less interference on the clients' requests. After data copy is complete, we remove half of the clients since they should be re-directed to another server. As one can see, the data migration takes about 8 seconds. During this procedure, unlike the Memcached experiments, data migration has little impact on the clients' requests, because it is performed from a non-leader server. Note that after mitigating bursty traffic, the performance of the two systems is different because they have different workloads: with PostMan, the server is processing full load with big packets; after data migration, the server is processing half load with small packets. Which one has better performance depends on the actual workload: in Figure 6 the system with PostMan has lower latency while in Figure 5 the system with PostMan has slightly higher latency.

Both the Memcached and the Paxos experiments have confirmed the effectiveness of PostMan: for services processing small packets, PostMan can quickly mitigate the long latency caused by unexpected bursty traffic, because it can offload the overhead of packet processing to helpers; data migration, on the other hand, takes much longer to achieve the same goal, and may further increase the latency during data migration when data migration competes for resource with processing clients' requests.

Capabilities of PostMan. To understand in what circumstances can PostMan help to mitigate bursty traffic, we measure how Memcached's and Paxos' tail latency grow with the load and how PostMan changes such trend.

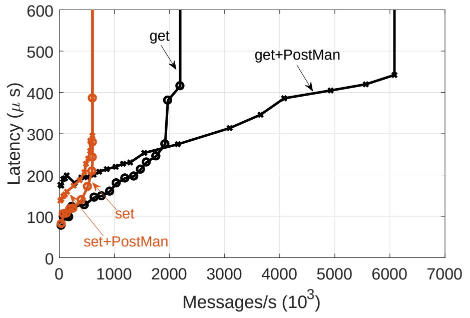


Fig. 7: The latency with different load for Memcached and Memcached + PostMan (64-byte payloads; PostMan enables up to five helper nodes).

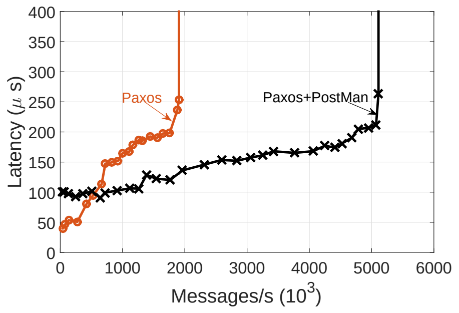


Fig. 8: The latency with different load for Paxos and Paxos + PostMan (64-byte payloads; PostMan enables up to six helper nodes).

As shown in Figure 7 and Figure 8, both Memcached get experiment and Paxos experiment show the same trend: when the load is low (i.e., lower than 2000K messages/s in Memcached and 500K messages/s in Paxos), using PostMan will actually introduce extra latency, because of the additional processing at the helper; when the load grows, the latency of the original systems will grow as well due to queuing delay and when these systems are close to saturation, their latency jumps, which is what happens when the service experiences bursty traffic. PostMan can offload their overhead of packet processing to helpers and thus can improve their maximum throughput, which reduces their queuing delay in a range of loads: for Memcached get experiments, PostMan can reduce the latency if the load is between 2000K and 6000K messages/s; for Paxos, PostMan can reduce its latency if the load is between 500K and 5000K messages/s. For ETC workload, when the bandwidth of helpees is saturated, PostMan can improve throughput by 1.6×; for USR workload, the throughput can be increased by 2.9×. Memcached set experiments do not benefit from PostMan, because as shown in our profiling, its bottleneck is lock contention, which has nothing to do with packet processing. This set of experiments show that PostMan is effective for a wide range of loads, but it does have its limits: that's why it is complementary to data migration, which does not have such limits but requires a longer time to be effective.

Real-world workloads. In the previous experiments, we evaluate the performance gain of PostMan with Memcached

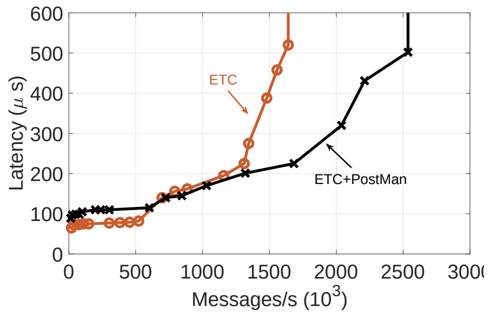


Fig. 9: The latency with different load for Memcached and Memcached + PostMan (ETC payload; PostMan enables up to six helper nodes).

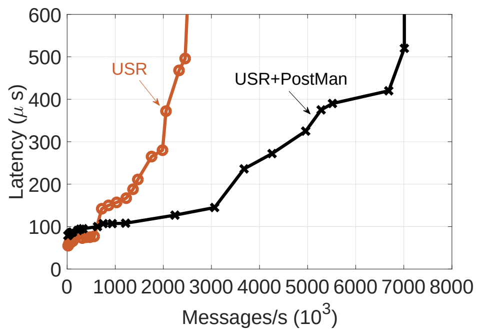


Fig. 10: The latency with different load for Memcached and Memcached + PostMan (USR payload; PostMan enables up to six helper nodes).

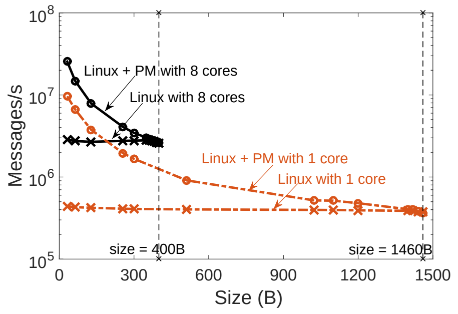


Fig. 11: The throughput with different payload sizes for Linux and Linux + PostMan (PostMan enables up to six helper nodes).

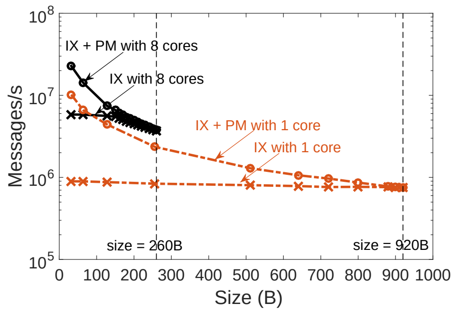


Fig. 12: The throughput with different payload sizes for IX and IX + PostMan (PostMan enables up to six helper nodes).

but only with laboratory-generated workloads. To evaluate whether PostMan is effective for realistic Memcached workload, we use *Mutilate* [55] as a load generator at the client side. *Mutilate* can coordinate a number of machines with multiple threads at the client side to generate Memcached workload based on the key-value distribution predetermined, while each machine can measure the average response latency and p99 tail latency for the requests. In this paper, we use mutilate to generate two kind of loads to represent two typical workloads reported by Facebook [17]: The ETC workload accounts for the largest proportion in Facebook, has 20B-70B keys, 1B-1kB value; The USR is a workload has two key size values, i.e., 16B (40%) and 21B (60%), and just one value size (2B). In these two workloads, each request is sent separately and the time gap of the requests is generated by a Generalized Pareto distribution with parameters Generalized Pareto with parameters $\sigma = 0$, $\delta = 16.0292$ and $k = 0.154971$, starting from the second request.

Figure 9 and Figure 10 show that both the experiments with the ETC workload and the experiments with the USR workload have the same trend as the Memecached and Paxos experiments. Combined with previous analysis, because PostMan can essentially reduce the queuing delay, both the ETC experiments and the USR experiments can benefit from PostMan in a range of loads: for ETC workload, PostMan can reduce the latency if the load is between 1600K and 2600K message/s; for USR workload, PostMan can reduce the latency if the load is between 2500K and 7200K message/s. For ETC workload, when the bandwidth of helpees is saturated, PostMan can improve throughput by 1.6×; for USR workload, the throughput can be increased by 2.9×. Apparently, PostMan is more effective for the USR workload, because the USR workload has averagely much smaller packets than the ETC workload, as well as there are lower proportion of set requests in USR work.

6.2 Effects of packet size

The capability of PostMan is affected by the packet size because PostMan's key idea of assembling packets naturally works well with smaller packets. To quantitatively understand how PostMan's capability is affected by this factor, we use the ping-pong microbenchmark to measure the throughput of PostMan, since as shown in Section 6.1, the maximal throughput of PostMan determines the range of loads in which PostMan might be helpful. We compare the throughput of PostMan to those of Linux and IX [20]. Since we fail to run an IX server with more than nine cores, we reduce the number of cores to eight in this set of experiments. Since IX shows it can outperform mTCP [19], another popular network stack, we do not further compare PostMan with mTCP.

Figure 11 shows the throughput of Linux under different packet sizes, with and without PostMan. As shown in this figure, when the server can utilize 8 cores for packet processing, PostMan can improve throughput when the payload size is less than 400 bytes. However, for CPU-intensive applications, this may not be a fair comparison because

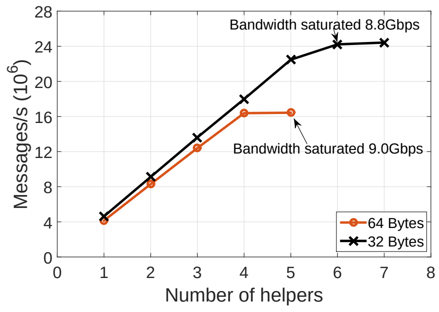


Fig. 13: The performance scales linearly when increasing the number of helpers nodes.

PostMan can reduce CPU utilization as well as improving throughput. Therefore, we also show the comparison when the server can utilize only one core for packet processing. In this case, PostMan can improve throughput when payload size is smaller than 1460 bytes.

Figure 12 shows the throughput of IX under different packet sizes, with and without PostMan. It shows a similar trend as the Linux experiment, though the turning points are smaller, 260B and 920B respectively. The benefit of PostMan still exists although becomes smaller than that in the Linux experiment. This is because IX, with an optimized networking stack, pays less overhead per packet compared to Linux.

Combining Figure 11 and Figure 12, one can see that Linux+Postman can even outperform IX when the packet size is small, despite the fact that the former approach does not require installing a new OS on all servers.

6.3 Performance of helper nodes

So far we have measured the performance gain at the server side. A natural question is how much resource we need to pay at the helper side to achieve such performance gain. To answer this question, we measure how much throughput a single helper can provide and whether PostMan scales with the number of helpers.

In this set of experiment, we use IX as the server side to ensure the server will not become the bottleneck. As Figure 13 shows, a single helper node can process about 9.6 million small messages (assembling 4.8 million requests and disassembling 4.8 million responses) per second. When increasing the number of helper nodes, the overall throughput of PostMan scales almost linearly, till the helpee's bandwidth is saturated.

Such results have demonstrated that PostMan does need a number of helper nodes to improve the throughput at the server side: this is as expected because PostMan offloads overhead instead of reducing overhead. Therefore, we expect a small-scale deployment of PostMan to help a few heavily loaded servers.

6.4 Effectiveness of enabling/disabling PostMan

To evaluate the effectiveness of enabling/disabling PostMan on demand, we set up a scenario where GET requests of Memcached arrive at the server side with 32B keys and 64B values. As shown in Figure 14, the load is low at first,

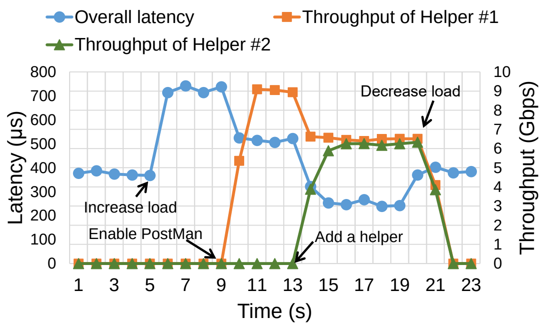


Fig. 14: PostMan is enabled/disabled on demand when the load of server becomes high/low.

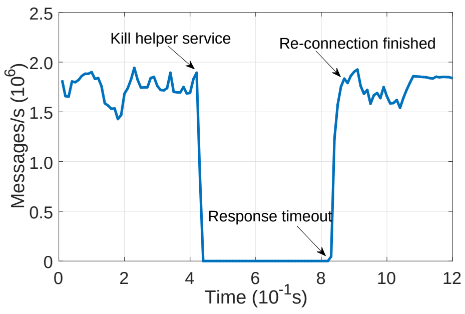


Fig. 15: The performance when PostMan recovers the connections by mapping them to another active helper.

which incurs an average latency of about $370\mu s$. Then we increase the load at the 5th time slot, making the average latency increase to about $700\mu s$. The server activates helper #1 and notifies the clients of disconnecting from the server. At the 9th time slot, the clients re-connect to helper #1. The re-connection involves 800 connections and finishes within $700ms$. Though the latency decreases to $500\mu s$, the load of the server is too high. At about the 13th time slot, helper #2 is added by the server. Then helper #1 disconnects half of the client connections and migrates them to the helper with lower throughput (helper #2) for load balancing. It takes about $250ms$ to finish migrating 400 connections. Note that we set a $3s$ delay (i.e., from the 5th time slot to the 8th time slot and from the 10th time slot to the 13th time slot) to display the experiment results (the change of latency and throughput caused by re-connecting and migration is too quick to be seen). At about the 20th time slot, the load decreases, PostMan is disabled and the clients directly connect to the server. The re-connection finishes in $450ms$, involving 800 connections. The results demonstrate that PostMan can be enabled/disabled on demand based on the fluctuating load.

6.5 Fault tolerance

To test whether PostMan can tolerate failures of helper nodes, we set up a simple scenario, in which two active helper nodes are connected to the server. A Linux client running on PostMan client library is performing request-reply communication to the server, with 10 threads and 1000 connections in total. To examine the failure recovery of helper nodes, we first let all clients connect to one helper, record the throughput (measured every $100ms$) of

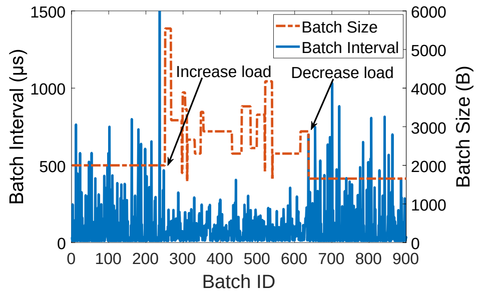


Fig. 16: Adaptively changing batch size and interval.

the clients and manually kill the connected helper node. As shown in Figure 15, the library waits until a timeout on receiving the reply message, and then reconnects to the other helper node. The migration takes about $400ms$ to recover all 1000 connections and restore to the previous rate.

Note that PostMan's correctness does not rely on the correctness of timeout: even if the timeout is inaccurate (i.e., a timeout is triggered when the previous helper node is still alive), PostMan can still guarantee all its properties because clients and servers will close the old connection and exchange necessary information when establishing a new connection (Section 4.2). This means in practice the developers can use a shorter timeout to improve availability. In this experiment, we simply use an arbitrary 1-second timeout to show that PostMan can function correctly despite failures.

6.6 Adaptive batching

To test the effectiveness of our adaptive batching algorithm, we change the load of our system and see how PostMan reacts. As shown in Figure 16, with the load increasing, the batch interval decreases, implying that the helper nodes batch packets more frequently. Although the size variation seems noisy, the batch sizes are mostly larger than those with low load. When the load decreases, the helper nodes can reduce the batch size and increase the batch interval. As a result, we can conclude that the batch size and batch interval are well adapted to fluctuating load.

7 Conclusion

In this paper, we present PostMan, a distributed service to mitigate load imbalance caused by bursty traffic, by offloading the overhead of packet processing from heavily-loaded servers and reducing data redundancy in packet headers. By batching small packets remotely and on demand, PostMan can utilize multiple nodes to help a heavily-loaded server when bursty traffic occurs and can minimize the overhead when there is no such bursty traffic. We design the helper network stack with DPDK and mTCP to improve packet processing efficiency and remove duplicated headers to reduce bandwidth consumption. To tolerate helper failures, PostMan can migrate connections across helpers. Because of their stateless nature, PostMan helpers are highly scalable as well. PostMan not only can be enabled when the load of the server becomes high, but also can scale well with the fluctuating load. Experiments with Memcached and Paxos show that, compared to data migration, PostMan can quickly mitigate the extra latency caused by bursty traffic. Furthermore, PostMan can improve the goodput of Memcached and Paxos by $3.3\times$ and $2.8\times$, respectively. When processing small packets, the throughput of PostMan even outperforms IX over $10\times$ at most with one core and increases linearly when the number of helper nodes grows. When a helper node fails, it only takes about $0.4s$ to reconnect 1000 connections and restore to the previous throughput.

References

[1] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The google file system," in *Proceedings of SOSP*, 2003.

[2] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop Distributed File System," in *Proceedings of MSST*, 2010.

[3] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," in *Proceedings of OSDI*, 2004.

[4] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A Distributed Storage System for Structured Data," in *Proceedings of OSDI*, 2006.

[5] "Apache HBASE," http://hbase.apache.org/.

[6] J. MacCormick, N. Murphy, M. Najork, C. A. Thekkath, and L. Zhou, "Boxwood: Abstractions as the Foundation for Storage Infrastructure," in *Proceedings of OSDI*, 2004.

[7] B. Calder, J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold, S. McKelvie, Y. Xu, S. Srivastav, J. Wu, H. Simitci, J. Haridas, C. Uddaraju, H. Khatri, A. Edwards, V. Bedekar, S. Mainali, R. Abbasi, A. Agarwal, M. F. u. Haq, M. I. u. Haq, D. Bhardwaj, S. Dayanand, A. Adusumilli, M. McNett, S. Sankaran, K. Manivannan, and L. Rigas, "Windows Azure Storage: a highly available cloud storage service with strong consistency," in *Proceedings of SOSP*, 2011.

[8] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford, "Spanner: Google's Globally-Distributed Database," in *Proceedings of OSDI*, 2012.

[9] E. B. Nightingale, J. Elson, J. Fan, O. Hofmann, J. Howell, and Y. Suzue, "Flat Datacenter Storage," in *Proceedings of OSDI*, 2012.

[10] D. Ongaro, S. M. Rumble, R. Stutsman, J. Ousterhout, and M. Rosenblum, "Fast Crash Recovery in RAMCloud," in *Proceedings of SOSP*, 2011.

[11] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's Highly Available Key-value Store," in *Proceedings of SOSP*, 2007.

[12] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. Li, M. Marchukov, D. Petrov, L. Puzar, Y. J. Song, and V. Venkataramani, "Tao: Facebook's distributed data store for the social graph," in *Proceedings of ATC*, 2013.

[13] "Glastonbury ticket website crashes," https://www.theguardian.com/music/2016/oct/09/glastonbury-ticket-website-crashes, 2016.

[14] "Macy's Web Site Buckles Under Heavy Traffic on Black Friday," http://fortune.com/2016/11/25/macys-black-traffic/, 2016.

[15] Y. Niu, B. Luo, F. Liu, J. Liu, and B. Li, "When hybrid cloud meets flash crowd: Towards cost-effective service provisioning," in *Proceedings of INFOCOM*, 2015.

[16] Y. Niu, F. Liu, X. Fei, and B. Li, "Handling flash deals with soft guarantee in hybrid cloud," in *Proceedings of INFOCOM*, 2017.

[17] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "Workload analysis of a large-scale key-value store," in *Proceedings of SIGMETRICS*, 2012.

[18] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab *et al.*, "Scaling memcache at facebook," in *Proceedings of NSDI*, 2013.

[19] E. Jeong, S. Wood, M. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park, "mTCP: a Highly Scalable User-level TCP Stack for Multicore Systems," in *Proceedings of NSDI*, 2014.

[20] A. Belay, G. Prekas, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion, "Ix: A protected dataplane operating system for high throughput and low latency," in *Proceedings of OSDI*, 2014.

[21] P. Jin, J. Guo, Y. Xiao, R. Shi, Y. Niu, F. Liu, C. Qian, and Y. Wang, "Postman: rapidly mitigating bursty traffic by offloading packet processing," in *Proceedings of USENIX ATC*, 2019, pp. 849–862.

[22] U. U. Hafeez, M. Wajahat, and A. Gandhi, "Elmem: Towards an elastic memcached system," in *Proceedings of ICDCS*, 2018.

[23] B. He, M. Yang, Z. Guo, R. Chen, B. Su, W. Lin, and L. Zhou, "Comet: batched stream processing for data intensive distributed computing," in *Proceedings of ACM symposium on Cloud computing*, 2010.

[24] B. Li, Z. Ruan, W. Xiao, Y. Lu, Y. Xiong, A. Putnam, E. Chen, and L. Zhang, "Kv-direct: high-performance in-memory key-value store with programmable nic," in *Proceedings of SOSP*, 2017.

[25] L. Mai, L. Rupprecht, A. Alim, P. Costa, M. Migliavacca, P. Pietzuch, and A. L. Wolf, "NetAgg: Using Middleboxes for Application-specific On-path Aggregation in Data Centres," in *Proceedings of CoNEXT*, 2014.

[26] "Introduction to Parallel I/O," https://www.olcf.ornl.gov/wp-content/uploads/2011/10/Fall_IO.pdf.

[27] "Dpdk (data plane development kit)," https://www.dpdk.org/.

[28] A. Belay, G. Prekas, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion, "Ix: A protected dataplane operating system for high throughput and low latency," in *Proceedings of OSDI*, 2014.

[29] S. Peter, J. Li, I. Zhang, D. R. K. Ports, D. Woos, A. Krishnamurthy, T. Anderson, and T. Roscoe, "Arrakis: The operating system is the control plane," in *Proceedings of OSDI*, 2014.

[30] L. Rizzo, "netmap: A Novel Framework for Fast Packet I/O," in *Proceedings of USENIX Security Symposium*, 2012.

[31] A. K. M. Kaminsky and D. G. Andersen, "Design guidelines for high performance rdma systems," in *Proceedings of ATC*, 2016.

[32] A. Kalia, M. Kaminsky, and D. G. Andersen, "Fasst: Fast, scalable and simple distributed transactions with two-sided (rdma) datagram rpcs," in *Proceedings of OSDI*, 2016.

[33] J. Shi, Y. Yao, R. Chen, H. Chen, and F. Li, "Fast and concurrent rdf queries with rdma-based distributed graph exploration," in *Proceedings of OSDI*, 2016.

[34] X. Wei, J. Shi, Y. Chen, R. Chen, and H. Chen, "Fast in-memory transaction processing using rdma and htm," in *Proceedings of SOPS*, 2015.

[35] Y. Chen, X. Wei, J. Shi, R. Chen, and H. Chen, "Fast and general distributed transactions using rdma and htm," in *Proceedings of EuroSys*, 2016.

[36] A. Kalia, M. Kaminsky, and D. G. Andersen, "Design guidelines for high performance rdma systems," in *Proceedings of ATC*, 2016.

[37] X. Lu, D. Shankar, S. Gugnani, and D. Panda, "High-Performance Design of Apache Spark with RDMA and Its Benefits on Various Workloads," in *Proceedings of IEEE International Conference on Big Data*, 2016.

[38] J. Liu, J. Wu, and D. K. Panda, "High performance rdma-based mpi implementation over infiniband," *Int. J. Parallel Program.*, vol. 32, no. 3, pp. 167–198, Jun. 2004.

[39] N. Islam, W. Rahman, X. Lu, and D. Panda, "High Performance Design for HDFS with Byte-Addressability of NVM and RDMA," in *Proceedings of ICS*, 2016.

[40] D. Shankar, X. Lu, N. Islam, W. Rahman, and D. Panda, "High-Performance Hybrid Key-Value Store on Modern Clusters with RDMA Interconnects and SSDs: Non-blocking Extensions, Designs, and Benefits," in *Proceedings of IPDPS*, 2016.

[41] W. Rahman, X. Lu, N. Islam, R. Rajachandrasekar, and D. Panda, "High-Performance Design of YARN MapReduce on Modern HPC Clusters with Lustre and RDMA," in *Proceedings of IPDPS*, 2015.

[42] W. Reese, "Nginx: the high-performance web server and reverse proxy," *Linux Journal*, vol. 2008, no. 173, p. 2, 2008.

[43] M.-C. Rosu and D. Rosu, "An evaluation of tcp splice benefits in web proxy servers," in *Proceedings of WWW*, 2002.

[44] C. Gunaratne, K. Christensen, and B. Nordman, "Managing energy consumption costs in desktop pcs and lan switches with proxying, split tcp connections, and scaling of link speed," *International Journal of Network Management*, vol. 15, no. 5, pp. 297–310, 2005.

[45] A. Pathak, Y. A. Wang, C. Huang, A. Greenberg, Y. C. Hu, R. Kern, J. Li, and K. W. Ross, "Measuring and evaluating tcp splitting for cloud services," in *Proceedings of International Conference on Passive and Active Measurement*, 2010.

[46] E. Y. Jeong, S. Woo, M. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park, "mtcp: A highly scalable user-level tcp stack for multicore systems," in *Proceedings of NSDI*, 2014.

[47] "Receive side scaling on intel network adapters," https://www.intel.com/content/www/us/en/support/network-and-i-o/ethernet-products/000006703.html.

[48] "consul," https://www.consul.io, 2020.

[49] "consul-template," https://learn.hashicorp.com/consul/developer-configuration/consul-template, 2020.

[50] "Memcached," http://memcached.org.

[51] L. Lamport, "Paxos Made Simple," *ACM SIGACT News (Distributed Computing Column)*, vol. 32, no. 4, pp. 51–58, Dec. 2001.

[52] J. Baker, C. Bond, J. C. Corbett, J. Furman, A. Khorlin, J. Larson, J.-M. Leon, Y. Li, A. Lloyd, and V. Yushprakh, "Megastore: Providing Scalable, Highly Available Storage for Interactive Services," in *Proceedings of CIDR*, 2011.

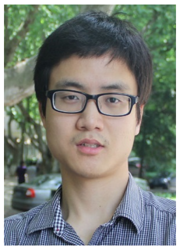
[53] "CloudLab," https://cloudlab.us.

[54] "memcache-mover," https://github.com/quipo/memcache-mover, 20017.

[55] "mutilate," https://github.com/leverich/mutilate, 2013.

Yipei Niu received his B.Eng. degree from Henan University, and M.Engr. degree from Huazhong University of Science and Technology. He is currently a Ph.D. student in the School of Computer Science and Technology, Huazhong University of Science and Technology, China. His research interests include cloud computing, container networking, serverless computing, and FPGA acceleration.

Panpan Jin received her B.S. and M.S. degrees in computer science and technology from Huazhong University of Science and Technology, Wuhan, China, in 2020. Her research interests include data center networking and Network function Virturlization.



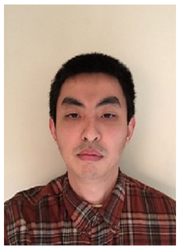
Jian Guo received his B.S. and Ph.D. degrees in the School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, China, in 2017. His research interests include data center networking and software-defined networking.

Yikai Xiao received his M.S. degree in the School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, China, in 2018. His research interests include data center networking and software-defined networking.

Rong Shi received his B.S. and M.S. in Computer Science and Engineering from University of Electronic Science and Technology of China in 2007 and 2011, and his Ph.D. in Computer Science and Engineering from Ohio State University in 2018. Currently, he is working at Facebook Inc. as a research scientist focusing on building systems to provide disaster recovery for the data warehouse. His research interests are in distributed systems, in particular fault tolerance and scalability.

Fangming Liu (S'08, M'11, SM'16) received the B.Eng. degree from the Tsinghua University, Beijing, and the Ph.D. degree from the Hong Kong University of Science and Technology, Hong Kong. He is currently a Full Professor with the Huazhong University of Science and Technology, Wuhan, China. His research interests include cloud computing and edge computing, datacenter and green computing, SDN/NFV/5G and applied ML/AI. He received the National Natural Science Fund (NSFC) for Excellent Young Scholars, and the National Program Special Support for Top-Notch Young Professionals. He is a recipient of the Best Paper Award of IEEE/ACM IWQoS 2019, ACM e-Energy 2018 and IEEE GLOBECOM 2011, the First Class Prize of Natural Science of Ministry of Education in China, as well as the Second Class Prize of National Natural Science Award in China.

Qian Chen is an Associate Professor in the Department of Computer Science and Engineering at University of California Santa Cruz. He primarily works on the fundamental problems of computer networks, systems, and security. He received his Ph.D. in Computer Science from University of Texas at Austin in 2013, M.Phil. from HKUST in 2008, and B.Sc. from Nanjing University in 2006. He received the NSF CAREER Award in 2018. He is a senior member of ACM and IEEE.



Yang Wang received the bachelor's and master's degrees in computer science and technology from Tsinghua University, in 2005 and 2008, respectively, and the doctorate degree in computer science from the University of Texas at Austin, in 2014. He is now an assistant professor in the Department of Computer Science and Engineering, Ohio State University. His current research interests include distributed systems, fault tolerance, and scalability.