PostMan: Rapidly Mitigating Bursty Traffic by Offloading Packet Processing

Panpan Jin[1], Jian Guo[1], Yikai Xiao[1], Rong Shi[2], Yipei Niu[1], Fangming Liu[1*], Chen Qian[3], Yang Wang[2]
[1]*National Engineering Research Center for Big Data Technology and System,*
Key Laboratory of Services Computing Technology and System, Ministry of Education,
School of Computer Science and Technology, Huazhong University of Science and Technology, China
[2]*The Ohio State University, USA*
[3]*University of California Santa Cruz, USA*

Abstract

Unexpected bursty traffic due to 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 others. To minimize the overhead, PostMan activates helpers on demand, only when bursty traffic is detected. To tolerate helper failures, PostMan can migrate connections across helpers and can ensure packet ordering despite such migration. 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.

1 Introduction

Modern distributed systems usually scale by partitioning data and assigning these partitions to different servers [4, 10, 11, 14, 15, 17, 33, 37, 41, 50]. In such an architecture, some servers may experience a higher load than others, creating a classic load imbalance problem [11, 16].

Many works have studied how to mitigate load imbalance by better data partitioning and placement strategies [4, 10, 11, 14, 16, 17, 41], 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 [16]. Unfortunately, unexpected bursty traffic is frequently reported in practice, for various reasons such as a sudden event drawing public attention on a social network [9] or a hot item on sale [18, 34, 39, 40].

This paper proposes PostMan, 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 median value size is 135 bytes [5, 38]; metadata servers, such as NameNode in HDFS [50], 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 2.4 Gbps with a CPU utilization of 800%. On the contrary, when processing 64KB packets, Linux can achieve a throughput of 9.1Gbps with a CPU utilization of only 220%. Newer networking stacks, such as mTCP [24] and IX [7], 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

*The corresponding author is Fangming Liu (fmliu@hust.edu.cn).

other components.

Motivated by this observation, PostMan incorporates a number of 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.

This approach brings several benefits: first, PostMan does not require the 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 capability 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 is able to achieve a considerable reduction in bandwidth consumption at the helpee.

Of course, PostMan has its own limitation: if load imbalance lasts long, PostMan will be more expensive than data migration because incorporating helpers incurs additional data transfer and 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 when the machine is less busy; 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, 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 three main components:

- 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 retransmit packets when necessary. While we find many applications already implement similar functionalities, we provide a library to those which do not.

- We present an adaptive batching mechanism to decide how many packets to assemble. It can increase batch size for higher throughput under heavy traffic and decrease 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 goodputs of Memcached and Paxos by 3.3× and 2.8×, respectively.

2 Related work

Load balancing. 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 [4, 10, 11, 14, 16, 17, 20, 41]; some of them can split, merge, and migrate existing data partitions online (e.g., Bigtable [11], RAMCloud [41] and ElMem [20]).

Despite the support from such mechanisms, how to mitigate load balancing caused by unexpected bursty traffic is still a challenge: since these mechanisms 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.

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 [21] batches the received data before batched stream processing at the server side. KV-direct [29] 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 a number of nodes to batch packets for other nodes. For example, Facebook has built mcrouter to batch packets for its Memcached service [38]. NetAgg aggregates traffic along network traffics for applications following a partition/aggregation pattern [35]. MPI has a collective I/O mode to batch I/Os from multiple processes before writing them to disks [36].

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 resource 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 [24] moves the TCP stack to the user space to reduce system call overhead and further improves performance by batching I/Os; DPDK [1] 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 [8] and Arrakis [43] design new operating systems to separate data transfer and access control to achieve both high speed and good protection; Netmap [46] improves networking performance by reducing memory allocation, system call, and data copy overhead; many works [12, 22, 25, 26, 30–32, 44, 48, 49, 51] exploit the RDMA technique and FPGA to improve networking performance.

Although these works have significantly improved the network performance, the performance gap between small and large packets persists (Table 1). Taking IX [8] 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 resource (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 [27].

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.

Others. The architecture of PostMan is similar to existing proxies (e.g., NGINX [45] and mcrouter [38]), 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 [19, 42, 47], which also deploys helper nodes in the network. However, their goals and internal mechanisms are totally different: split TCP is designed to reduce latency in a network with large round-trip delays, by letting helper nodes send *acks* 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 *acks* to the sender, until the helpers receive *acks* from the helpee (see Section 4.2).

	64 bytes	64KB
10Gb Linux	2.4Gpbs	9.1Gbps
10Gb IX [8]	5.0Gbps	9Gbps

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

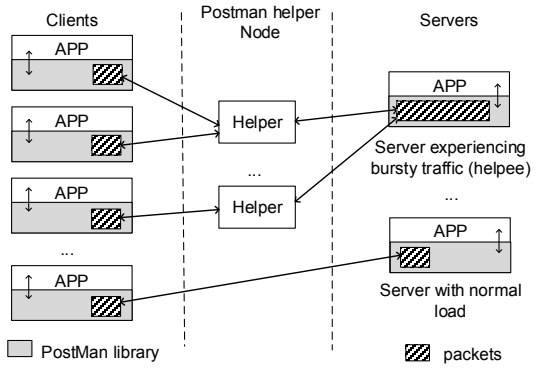

Figure 1: Overview of PostMan.

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 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 packet: 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,

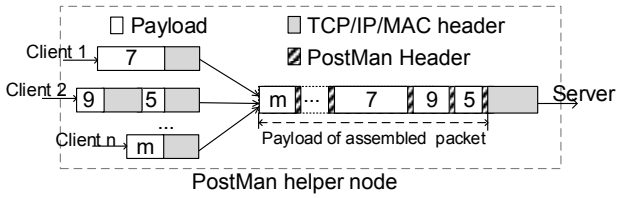

Figure 2: PostMan assembles packets from different clients by using a short header for each payload.

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)

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 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.

On 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.

4.2 PostMan library

A traditional networking library provides a number of APIs, such as *bind*, *connect*, *send*, and *recv* to the application. PostMan library provides a few additional ones: *pm_connect* allows a client to create a connection to a helper node; *compose* and *decompose* assemble and 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 5.1) 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. The helper node will connect to the corresponding helpee, if there is no connection yet, and forward this 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. 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 and process them. On the opposite direction, when a server sends replies, it should buffer multiple replies 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, one can easily verify that these properties hold when there is no migration of connections. However, PostMan may migrate connections for several reasons: if a client is connecting directly to a server and finds the latency is high, it will call *pm_connect* to migrate its connection to a helper; when a helper is heavily loaded, PostMan will instruct its clients to migrate to other helpers; finally if a helper fails, its clients must migrate to other helpers as well. 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 acknowledgement 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 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[1] will buffer packets until it receives *acks* from receivers, like 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.

[1] 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.

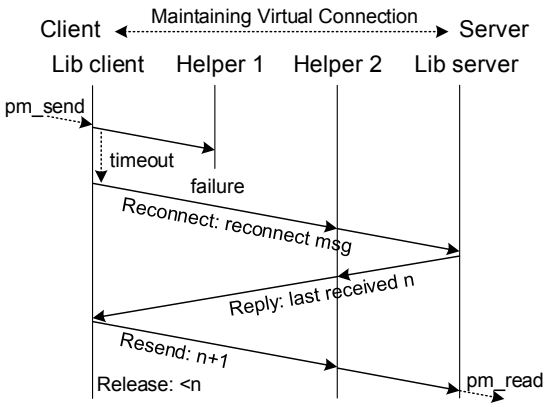

Figure 3: Maintaining a virtual connection by redirecting and re-sending requests when a helper node fails.

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 3, 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.

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) and should assemble 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 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

Algorithm 1 Adaptive batching algorithm

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

procedure Update S and T
  $S_l \leftarrow 0.75S$
  $S_u \leftarrow 1.25S$
  if $(s < S_l \vee s > S_u) \wedge (s \geq 1500)$ **then**
    $S \leftarrow s$
  end if
  $T_l \leftarrow 0.5T$
  $T_u \leftarrow 1.5T$
  if $(t < T_l \vee t > T_u) \wedge (t \geq 10\mu s)$ **then**
    $T \leftarrow t$
  end if
end procedure

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 [1], 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 [23] 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) [3] 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 Scalability and 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.

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 picks up one with a low load; 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 standard load-balancing technique: each helper monitors its own CPU and network utilization; when a client establishes a connection, either for a new connection or for re-connecting, it randomly chooses a number of helpers; each helper will reply with its resource utilization, so that the client can choose the helper with the lowest utilization; 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 the clients and the 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. 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.

Therefore, the current implementation of PostMan uses a hybrid approach: a client will decide to enable helpers (i.e., the client disconnects from the server and connects to a helper) if it detects its perceived latency is higher than SLA—this could minimize the overhead at the server side when the server is busy; a server will decide to disable helpers when its throughput becomes low—the overhead is fine in this case since the server is not too busy.

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.

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 [2]. It is used widely in the data centers to cache data to speed up the lookups of frequently accessed data. As reported in [5], 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 [28]. 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 [6], Windows Azure [10] and Spanner [14], 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 [8] 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 [13] 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 [8], 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

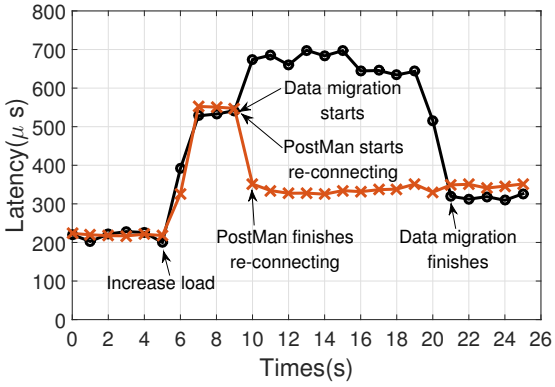

Figure 4: Mitigating bursty traffic in Memcached (PostMan enables two helpers).

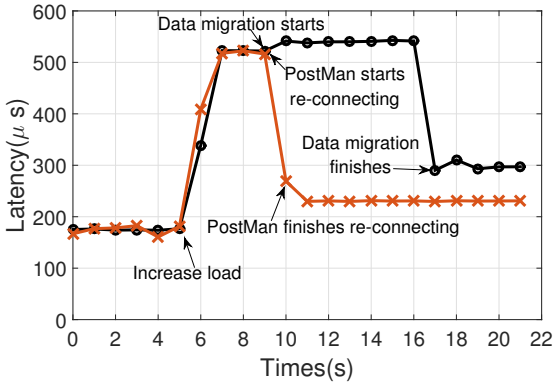

Figure 5: Mitigating bursty traffic in Paxos (PostMan enables two helpers).

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 [7].

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 p99 latency of Memcached using a read workload with 32B keys and 64B values. As shown in Figure 4, we first use a light load, which incurs a latency of 200 µs, till time 5. Then we increase the load drastically, which increases the latency to more than 500 µs. At about time 9, we enable PostMan, which asks clients to re-connect to helpers. Such re-connection involves 660 client connections and finishes within 550 ms. Afterwards, the latency is reduced to around 300 µs.

As a comparison, we emulate the data migration approach by assuming 50% of the clients are accessing 10% of the data (i.e., 6.4GB) in the Memcached server. Therefore, we start a thread in the Memcached server to copy the data to another server at time 9. Normally such a thread needs to access the internal data structure of Memcached, which may incur additional CPU overhead and may contend with the client's requests. Our emulation avoids such overhead by letting the migration thread copy dummy data to another server, which is in favor of the data migration approach. 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 request, so the content of the request does not matter). As shown in Figure 5, we first use a light load, which incurs a latency of 200 µs, till time 5. Then we increase the load, which increases the latency to about 500 µs. At about time 9, we enable PostMan, which asks clients to re-connect to helpers. Such re-connection involves 960 client connections and finishes within 750 ms. Afterwards, the latency is reduced to around 220 µs.

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 time 9. 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 5 the system with PostMan has lower latency while in Figure 4 the system with PostMan has slightly higher latency.

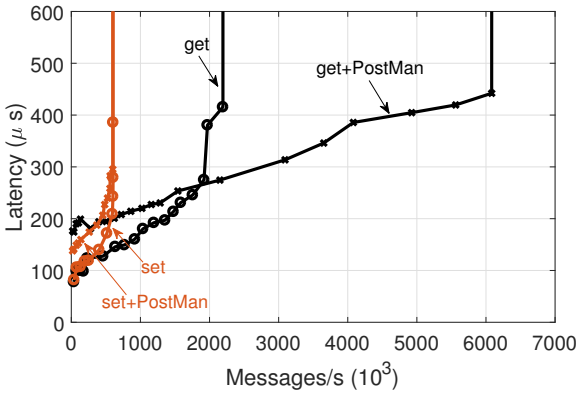

Figure 6: The latency with different load for Memcached and Memcached + PostMan (64-byte payloads; PostMan enables up to five helper nodes).

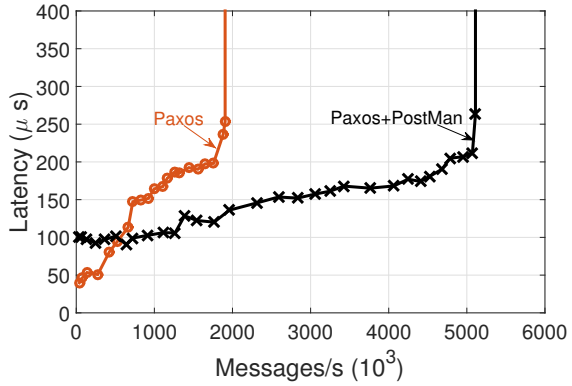

Figure 7: The latency with different load for Paxos and Paxos + PostMan (64-byte payloads; PostMan enables up to six helper nodes).

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.

As shown in Figure 6 and Figure 7, 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

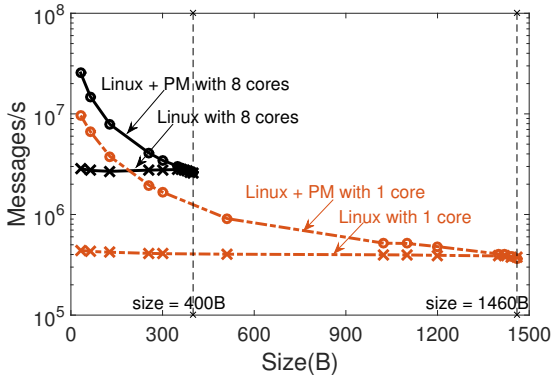

Figure 8: The throughput with different payload sizes for Linux and Linux + PostMan (PostMan enables up to six helper nodes).

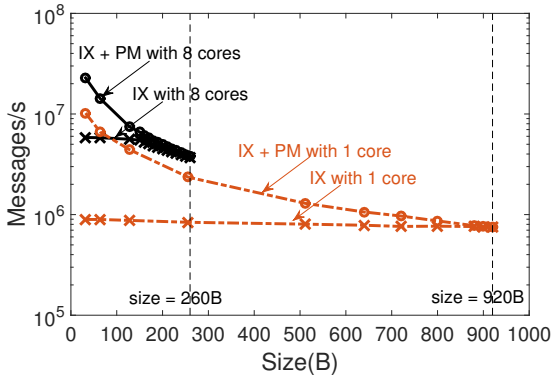

Figure 9: The throughput with different payload sizes for IX and IX + PostMan (PostMan enables up to six helper nodes).

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. 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.

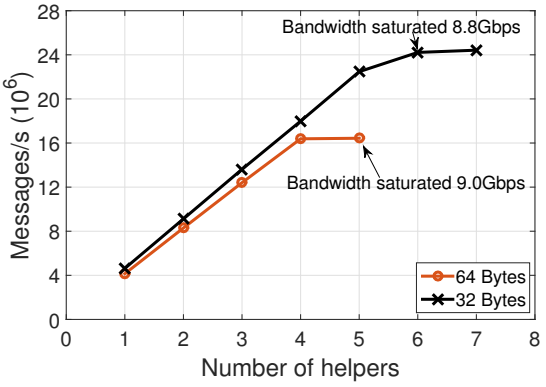

Figure 10: The performance scale linearly when increasing the number of helpers nodes.

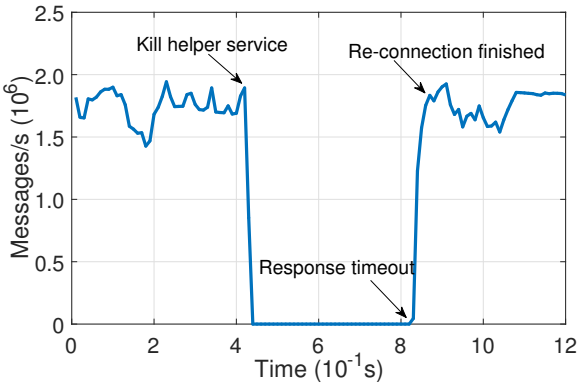

Figure 11: The performance when PostMan recovers the connections by mapping them to another active helper.

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 [7]. 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 [24], another popular network stack, we do not further compare PostMan with mTCP.

Figure 8 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 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 9 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.

Comparing Figure 8 and Figure 9, 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 10 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 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 the clients and manually kill the connected helper node. As shown in Figure 11, the library waits until a timeout on receiving the reply message, and then reconnects to the other helper node. The re-connection takes about

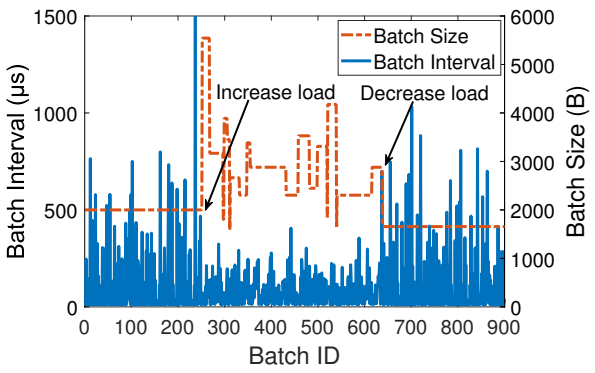

Figure 12: Adaptively changing batch size and interval.

0.4s to recover all 1000 connections. After that, the message exchange reaches 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.

6.5 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 12, 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. Experiments with Memcached and Paxos show that, compared to data migration, PostMan can quickly mitigate the extra latency caused by bursty traffic.

8 Acknowledgment

We thank our shepherd Boris Grot and other anonymous reviewers for their insightful comments. This work was supported in part by the NSFC under Grant 61761136014 (and 392046569 of NSFC-DFG) and 61722206 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.

References

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

[2] Memcached. http://memcached.org.

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

[4] Apache HBASE. http://hbase.apache.org/.

[5] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of SIGMETRICS*, 2012.

[6] Jason Baker, Chris Bond, James C. Corbett, JJ Furman, Andrey Khorlin, James Larson, Jean-Michel Leon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. Megastore: Providing Scalable, Highly Available Storage for Interactive Services. In *Proceedings of CIDR*, 2011.

[7] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. Ix: A protected dataplane operating system for high throughput and low latency. In *Proceedings of OSDI*, 2014.

[8] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. Ix: A protected dataplane operating system for high throughput and low latency. In *Proceedings of OSDI*, 2014.

[9] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, Mark Marchukov, Dmitri Petrov, Lovro Puzar, Yee Jiun Song, and Venkat

Venkataramani. Tao: Facebook's distributed data store for the social graph. In *Proceedings of ATC*, 2013.

[10] Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam McKelvie, Yikang Xu, Shashwat Srivastav, Jiesheng Wu, Huseyin Simitci, Jaidev Haridas, Chakravarthy Uddaraju, Hemal Khatri, Andrew Edwards, Vaman Bedekar, Shane Mainali, Rafay Abbasi, Arpit Agarwal, Mian Fahim ul Haq, Muhammad Ikram ul Haq, Deepali Bhardwaj, Sowmya Dayanand, Anitha Adusumilli, Marvin McNett, Sriram Sankaran, Kavitha Manivannan, and Leonidas Rigas. Windows Azure Storage: a highly available cloud storage service with strong consistency. In *Proceedings of SOSP*, 2011.

[11] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A Distributed Storage System for Structured Data. In *Proceedings of OSDI*, 2006.

[12] Yanzhe Chen, Xingda Wei, Jiaxin Shi, Rong Chen, and Haibo Chen. Fast and general distributed transactions using rdma and htm. In *Proceedings of EuroSys*, 2016.

[13] CloudLab. https://cloudlab.us.

[14] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's Globally-Distributed Database. In *Proceedings of OSDI*, 2012.

[15] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proceedings of OSDI*, 2004.

[16] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's Highly Available Key-value Store. In *Proceedings of SOSP*, 2007.

[17] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *Proceedings of SOSP*, 2003.

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

[19] Chamara Gunaratne, Ken Christensen, and Bruce 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*, 15(5):297–310, 2005.

[20] Ubaid Ullah Hafeez, Muhammad Wajahat, and Anshul Gandhi. Elmem: Towards an elastic memcached system. In *Proceedings of ICDCS*, 2018.

[21] Bingsheng He, Mao Yang, Zhenyu Guo, Rishan Chen, Bing Su, Wei Lin, and Lidong Zhou. Comet: batched stream processing for data intensive distributed computing. In *Proceedings of ACM symposium on Cloud computing*, 2010.

[22] 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.

[23] Eun Young Jeong, Shinae Woo, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. mtcp: A highly scalable user-level tcp stack for multicore systems. In *Proceedings of NSDI*, 2014.

[24] EunYoung Jeong, Shinae Wood, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. mTCP: a Highly Scalable User-level TCP Stack for Multicore Systems. In *Proceedings of NSDI*, 2014.

[25] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Design guidelines for high performance rdma systems. In *Proceedings of ATC*, 2016.

[26] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Fasst: Fast, scalable and simple distributed transactions with two-sided (rdma) datagram rpcs. In *Proceedings of OSDI*, 2016.

[27] Anuj Kalia Michael Kaminsky and David G Andersen. Design guidelines for high performance rdma systems. In *Proceedings of ATC*, 2016.

[28] Leslie Lamport. Paxos Made Simple. *ACM SIGACT News (Distributed Computing Column)*, 32(4):51–58, December 2001.

[29] Bojie Li, Zhenyuan Ruan, Wencong Xiao, Yuanwei Lu, Yongqiang Xiong, Andrew Putnam, Enhong Chen, and Lintao Zhang. Kv-direct: high-performance in-memory key-value store with programmable nic. In *Proceedings of SOSP*, 2017.

[30] Xiaoyao Li, Xiuxiu Wang, Fangming Liu, and Hong Xu. Dhl: Enabling flexible software network functions with fpga acceleration. In *Proceedings of ICDCS*, 2018.

[31] Jiuxing Liu, Jiesheng Wu, and Dhabaleswar K. Panda. High performance rdma-based mpi implementation over infiniband. *Int. J. Parallel Program.*, 32(3):167–198, June 2004.

[32] 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.

[33] John MacCormick, Nick Murphy, Marc Najork, Chandramohan A. Thekkath, and Lidong Zhou. Boxwood: Abstractions as the Foundation for Storage Infrastructure. In *Proceedings of OSDI*, 2004.

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

[35] Luo Mai, Lukas Rupprecht, Abdul Alim, Paolo Costa, Matteo Migliavacca, Peter Pietzuch, and Alexander L. Wolf. NetAgg: Using Middleboxes for Application-specific On-path Aggregation in Data Centres. In *Proceedings of CoNEXT*, 2014.

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

[37] Edmund B. Nightingale, Jeremy Elson, Jinliang Fan, Owen Hofmann, Jon Howell, and Yutaka Suzue. Flat Datacenter Storage. In *Proceedings of OSDI*, 2012.

[38] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, et al. Scaling memcache at facebook. In *Proceedings of NSDI*, 2013.

[39] Yipei Niu, Fangming Liu, Xincai Fei, and Bo Li. Handling flash deals with soft guarantee in hybrid cloud. In *Proceedings of INFOCOM*, 2017.

[40] Yipei Niu, Bin Luo, Fangming Liu, Jiangchuan Liu, and Bo Li. When hybrid cloud meets flash crowd: Towards cost-effective service provisioning. In *Proceedings of INFOCOM*, 2015.

[41] Diego Ongaro, Stephen M. Rumble, Ryan Stutsman, John Ousterhout, and Mendel Rosenblum. Fast Crash Recovery in RAMCloud. In *Proceedings of SOSP*, 2011.

[42] Abhinav Pathak, Y. Angela Wang, Cheng Huang, Albert Greenberg, Y. Charlie Hu, Randy Kern, Jin Li, and Keith W. Ross. Measuring and evaluating tcp splitting for cloud services. In *Proceedings of International Conference on Passive and Active Measurement*, 2010.

[43] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. Arrakis: The operating system is the control plane. In *Proceedings of OSDI*, 2014.

[44] 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.

[45] Will Reese. Nginx: the high-performance web server and reverse proxy. *Linux Journal*, 2008(173):2, 2008.

[46] Luigi Rizzo. netmap: A Novel Framework for Fast Packet I/O. In *Proceedings of USENIX Security Symposium*, 2012.

[47] Marcel-Catalin Rosu and Daniela Rosu. An evaluation of tcp splice benefits in web proxy servers. In *Proceedings of WWW*, 2002.

[48] 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.

[49] Jiaxin Shi, Youyang Yao, Rong Chen, Haibo Chen, and Feifei Li. Fast and concurrent rdf queries with rdma-based distributed graph exploration. In *Proceedings of OSDI*, 2016.

[50] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The Hadoop Distributed File System. In *Proceedings of MSST*, 2010.

[51] Xingda Wei, Jiaxin Shi, Yanzhe Chen, Rong Chen, and Haibo Chen. Fast in-memory transaction processing using rdma and htm. In *Proceedings of SOPS*, 2015.