Janus: A User-Level TCP Stack for Processing 40 Million Concurrent TCP Connections

Chao Zheng*†, Qi Tang§, Qiuwen Lu*, Jie Li*†, Zhou Zhou* and Qinyun Liu*
*Institute of Information Engineering, Chinese Academy of Sciences
†School of Cyber Security, University of Chinese Academy of Sciences
‡Tencent Inc.
Email:{zhengchao}@iie.ac.cn

***Abstract*—C10M is an Internet scalability problem regarding how to handle 10 million simultaneous TCP connections on a web server. Although kernel- and user-level approaches have been proposed to increase TCP stack scalability on multicore systems, C10M is still an open problem.**

In this paper we present Janus, a high-performance user-level TCP stack that focuses on serving massive TCP connections. In addition to adopting well-known techniques, our design (1) separates packet I/O cores from TCP processing cores to achieve high scalability and flexibility on a multicore system and (2) lets each application run as a per-connection coroutine together with a packet processing loop, which greatly improves cache affinity and saves memory. We demonstrate that Janus can accept 1.86 million new connections per second while maintaining 40 million concurrent connections and significantly outperforms Linux and state-of-the-art user-space network stacks in both throughput and connection concurrency.

I. Introduction

With the proliferation of the Internet, websites in the Alexa top 100 have huge numbers of users. Furthermore, emerging protocols such as HTTP/2 [4] and websocket [9] show a trend of using persistent TCP connections and thereby improving response times. As a result, web servers must serve massive numbers of concurrent TCP connections, especially web proxies, CDN (Content Delivery Network) cache servers, and so on. A decade after the C10k problem was solved, Robert Graham [10] posed the C10M challenge: how to handle 10,000,000 simultaneous TCP connections on a web server. To achieve C10M scalability, a network stack implementation must support fast packet processing, multi-core scalability, and resource efficiency. For the rigorous standard, the C10M is still an open problem. As far as we know, no approach to date has simultaneously met the requirements of concurrency and an acceptable new connection establishment rate. For example, the Migratory Data Server [20] can handle 12 million long connections, but with a very low message rate of sending each connection a 512-byte message every minute, and StackMap [28] has achieved a 200 μs latency, but the number of concurrent connections is limited to 100.

Qi Tang's work was done while he was a graduate student at IIE. Qinyun Liu is the corresponding author.

In this paper, we present Janus, a high-performance, user-level TCP stack that focuses on serving a massive number of TCP connections. In addition to adopting well-known techniques, such as the user-level stack, with nothing shared, no copying, and batching, Janus implements two distinct approaches to achieve 40 million concurrent connections:

- Janus separates packet I/O cores from TCP processing cores on a multicore system to achieve high scalability and flexibility. Through a specialized TCP load balancing algorithm called the consistent stream balancer (CSB), the workloads of the TCP processing cores can be adjusted dynamically.
- The application runs as a per-connection coroutine in the TCP processing thread to secure data locality, which also reduces per-connection memory usage.

In our benchmark tests on an x86 server with 8 CPU cores and 64B messages, we show that Janus' new connection rate outperforms state-of-the-art user-level stacks: it has 3.9× the connection rate of the Linux networking stack (kernel 3.10.0) and 1.8× that of mTCP [14]. With 10 CPU cores and 1kB messages, Janus can simultaneously achieve the performance shown in Table I, which meets the concurrency and new connection rate requirements of C10M.

TABLE I: C10M requirements and Janus's performance with 10 CPU cores and 1kB messages.

	C10M	**Janus**
Concurrent Connections	10,000,000	40,000,000
New Connections	1,000,000/s	1,670,000/s
Bandwidth	10 gigabits/s	20 gigabits/s
Packets Per Second	10,000,000/s	7,600,000/s
Latency[1]	10μs	230μs
Jitter	10μs	18μs

II. Related Work

In this section, we discuss current research on high-performance TCP I/O. This research falls into two areas:

[1]The latency of connection establishment three-way handshake.

optimizing the traditional kernel and bypassing the kernel.

Optimization of the traditional kernel has the greatest generalizability with respect to various protocol features. In addition, some approaches also retain compatibility with existing software, for the modifications occur beneath the BSD socket API. Recent studies have proposed various solutions to address inefficiencies in multicore systems, including the lack of connection locality [11], [16], [23], a shared file descriptor space [17], inefficient packet processing [28], and heavy system call overhead [11].

Although these proposals have addressed several shortcomings in the kernel stack, there are system call and context switching overheads that the traditional kernel cannot avoid. Moreover, recently emerging high-speed packet I/O frameworks such as DPDK [1], netmap [26], and PF_RING [22] provide unprecedented network performance for applications. Therefore, separating networking stacks from the traditional kernel has become more attractive, as this eliminates the previously mentioned overheads. For example, IX [3] and Arrakis [24] bypass the traditional kernel with a newly designed data plane running at the kernel level, thereby achieving high I/O performance. MultiStack [12] endeavors to ease the deployment of new protocols, but scalability for massive TCP connections is not considered. SandStorm [18] attempts to build highly specialized, application-specific stacks to achieve high throughput, but it is clearly not designed for massive connections—the testing ended at 80 concurrent connections. mTCP [14] addresses the performance of short-lived TCP connections on multicore systems. It implements a per-core management structure at the user level and uses packet batching to amortize costs. As a result, mTCP can accept 0.8 million short-lived TCP connections per second and achieves 6Gbps throughput.

While these approaches eliminate context-switching overheads, taken alone they do not eliminate the difficult tradeoffs between concurrency and user friendliness. More specifically, the scalability of creating and maintaining millions of concurrent connections at the same time is barely considered. Indeed, short TCP connections and long connections can coexist on the same server, e.g., a chat server [25]. Moreover, on these approaches, optimization of application interfaces is limited to callbacks and event queues, which are either difficult to use or degenerate the cache affinity.

III. Janus's Architecture

The main goal of Janus is to build a highly scalable TCP stack on multicore systems that can handle over 40 million concurrent connections and at the same time provide user-friendly APIs. As shown in Figure 1, the packet I/O runs on several independent threads that send and receive packets via a DPDK Poll Mode Driver (PMD), and TCP flows are dispatched to the remaining cores, which execute the TCP stacks and the server side applications running on top of them. The TCP processing cores run all stages that are needed to enqueue and dequeue rings to completion, interleaving protocol processing and application coroutines at well-defined transition points. The load balancing algorithm CSB dispatches flows to the different cores in such a way that a single TCP flow will always be handled by the same core. The control plane monitors TCP performance and adjusts the workload between TCP processing threads by manipulating the CSB. Packet buffers are located in shared memory, so access by multiple processes is allowed.

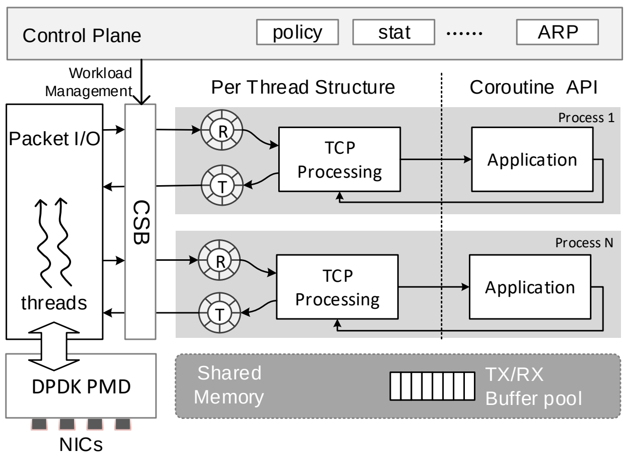

Fig. 1: Janus Design Overview.

In this section, we introduce three key components that enable Janus's scalability for 40 million concurrent connections.

A. Packet I/O

Several technologies have been developed for achieving fast packet I/O on commodity hardware, such as DPDK [1], netmap [26], and PF_RING [22]. Of these technologies, the authors chose DPDK as the I/O infrastructure for Janus. DPDK's performance and stability as a fast user-space packet I/O engine have been proven by researchers and companies. Compared to netmap and PF_RING, DPDK supports more hardware and has a thriving community of users.

We crafted the original DPDK [1] to pipeline, so that with Janus it is possible to separate packet I/O. Janus separates packet I/O from TCP processing to obtain three advantages. First, I/O threads can distribute packets based on the destination port, which provides isolation between different applications, e.g., running a web server and a NoSQL store in different processes on the same host. Second, with CSB, separate I/O threads can flexibly distribute packets to processing threads, unlike multi-queue NIC receive-side scaling (RSS [8]), which cannot adjust dynamically. Third, users can easily enforce security policies and debugging, e.g., dropping unusual packets or tracking specific flows. The number of packet I/O threads can be increased to handle more packets, so packet I/O will not become a bottleneck.

B. Scheduling

For efficient TCP processing, we let the application and TCP stack run in the same main loop so that they both achieve better data locality.

Most modern web applications serve multiple clients with each server thread through either (1) **nonblocking I/O** or (2) **asynchronous I/O**. In scheme (1), applications acquire file descriptor readiness notifications through `select` or `epoll`. mTCP [14] implements a similar interface at the user level. This causes overhead in terms of extra memory copies and context switches, which would be considerable for 40 million concurrent connections. Scheme (2) is at a higher level of abstraction than nonblocking I/O and requires breaking applications into several I/O-triggered event handlers. IX [3] and SandStorm [18] both adopt this model to avoid memory copies. Although asynchronous I/O is efficient, its programming is complicated and unnatural [27].

To provide applications with a concise interface, Janus introduces the concept of coroutines to implement blocking I/O. Coroutines can yield the CPU to other eligible tasks if a given condition does not hold, e.g., if a send request cannot be promptly completed because there are no available send buffers. When rescheduled, tasks resume exactly where they left off; applications are not aware of context switching; and they can be programmed with a simpler blocking-style API.

The default stack size for a Linux program is 8MB. With millions of concurrent connections, the memory consumption that would be required for providing a separate stack for each coroutine is far too huge. Fortunately, we've noticed that web applications perform I/O operations with a lower stack depth and that the exact stack size for an inactive coroutine is small. For this reason, Janus uses a shared stack model to achieve a high degree of concurrency, with stacks copied into and out of an 8-MB per-thread runtime stack. When a coroutine yields the CPU, its current stack is saved in dynamically allocated memory.

Programming languages like Go and Python have native support for coroutines. Janus uses C and assembly instructions (ASM) to save and restore the application context; context switching between coroutines is light-weight. With a 1KB stack size, our coroutines cost 64 nanoseconds for one resume/yield, which is twice as much as a function call (33 nanoseconds). This will be further evaluated in Section V. This cost can also be further amortized through batching techniques like Linux's New API (NAPI) [21].

In short, the coroutine scheduler promises to achieve cache affinity and still provide a convenient API.

C. Consistent Stream Balancer

As Janus separates packet I/O from TCP/application processing cores, load balancing is crucial for the scalability of the data plane. The original DPDK and recent approaches [11], [17] use RSS to distribute packets, but a fixed hash function is not very flexible. We designed a fine-grained load balancing algorithm, the CSB, for dispatching flows to different working cores based on consistent hashing [15] and a Bloom filter [5]. As shown in Fig. 2, there are three roles in the packet distribution process: packet I/O threads, TCP processing threads, and a control plane thread. The CSB allows the control plane to perform a live reconfiguration of the load balancing hash function if the connection distribution to the TCP processing threads is unbalanced, and it still ensures that a single TCP connection will always be handled by the same core. As in the conventional consistent hash algorithm, the CSB works by associating each thread ID (`tid`[2]) with a number of randomly chosen points on the unit circle. Given a key, i.e., the 4-tuple associated with a packet[3], the CSB hashes the key to a position on the unit circle, proceeds from there in a clockwise direction along the circle until it finds the first neighboring point, and it returns the `tid` associated with that point. The probability that a `tid` is chosen is equal to the proportion of its related points. The control plane keeps track of the load across TCP threads by monitoring their receiving queue size, and if any thread is unbalanced, its workload can be adjusted by adding/deleting points associated with that thread to/from the circle. At this point the problem becomes complicated, because resizing the consistent hash may remap an old connection to a different thread, and the old connection will be terminated because it is not receiving any subsequent packets.

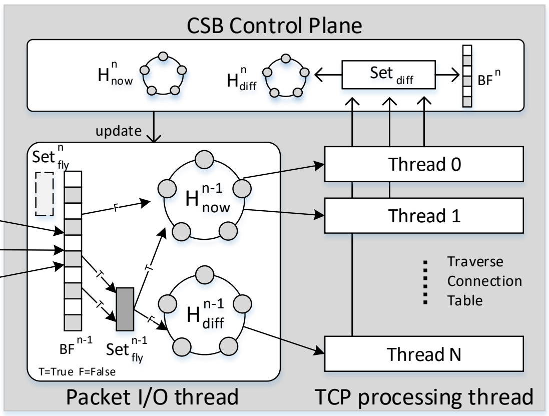

Fig. 2: The CSB Architecture. The control plane is under the *n*th update, and the packet I/O plane distributes packets with the *n-1*th data structure.

To address this problem, the CSB has two consistent hash data structures, H_{now} and H_{diff}, a Bloom filter BF, and a set Set_{fly} (for flying connection) to determine which hash should be used for a packet. The nth update of CSB is defined as follows:

[2]Each TCP processing thread is numbered with a value from 0 to n-1, where n is the number of TCP processing threads

[3]The 4-tuple for a packet refers to a remote IP, a remote port, a local IP, and a local port.

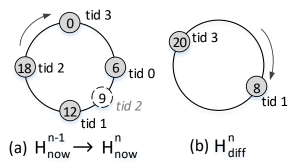

Fig. 3: A CSB sample

`tid`	points	keys
0	6	1,3,4
1	12	7,8,10,11,12
2	18	13,15
3	0	20,21,22

TABLE II: A sample connection table

Step 1. If the control plane detects an imbalance, it adds or deletes points in the current hash function H_{now}^{n-1} to obtain a new hash function H_{now}^{n}, just like the ordinary consistent hash algorithm.

Step 2. The control plane notifies the packet I/O thread to use H_{now}^{n} to distribute TCP SYN packets and record their 4-tuples in the set Set_{fly}^{n}. The packet I/O thread will determine whether a packet belongs to a flying connection with Set_{fly}^{n}, and if so, it will be distributed with H_{now}^{n}. This allows the CSB to adjust the workload without halting the distribution of connections as it ensures the consistency of flying connections.

Step 3. The control plane notifies each TCP processing thread (`tid`=i) that it should traverse each key of the thread's connection table, $Table_i$. If $H_{now}^{n}(key) \neq i$, then it sets $Add(BF^n, key)$.

Step 4. Each TCP processing thread traverses $Table_i$ again, and if $Query(BF^n, key)$ is positive, then (key, i) is added to the set Set_{diff}. In this way, each existing connection that is affected by H_{now}^{n} or is misidentified as affected due to a false positive from the Bloom filter BF^n is collected in Set_{diff}.

Step 5. The control plane sorts the keys in Set_{diff} in ascending order. Sequential keys sharing the same `tid` can be associated with the largest key, and this is stored in H_{diff}^{n}. Thus, H_{diff}^{n} records Set_{diff} in a compact manner.

Step 6. H_{now}^{n-1} and H_{diff}^{n-1} are replaced with H_{now}^{n} and H_{diff}^{n}. The packet I/O thread shrinks the set Set_{fly}^{n} by eq. 1. In practice, Set_{fly}^{n} stores about 100 keys for a 1000- milliseconds update, so this is not a big burden.

$$\begin{aligned} Set_{fly}^{n} = \{key \mid\ & key \in Set_{fly}^{n} \\ & \wedge Query(BF^n, key)\ is\ positive \\ & \wedge H_{now}^{n}(key) \neq H_{diff}^{n}(key)\} \end{aligned} \tag{1}$$

We use the example in Figure 3 and Table II to illustrate how the CSB works. There are 4 threads, and the key space ranges from 0 to 23. Connections are distributed using the H_{now}^{n-1} in Figure 3(a) (shaded circles) after nth update. At the moment (Steps 1 and 2), thread 1 is processing 5 connections, which is more than other threads are processing, so the control plane decides to place a new point at position 9 to port new connections to thread 2, producing H_{now}^{n}, which is Figure 3(a) in its entirety. Given H_{now}^{n}, keys 7 and 8 will be ported to thread 2, so (Step 3; we will further discuss Step 2 below) the TCP thread sets the Bloom filter to reflect this, i.e., it sets $Add(BF^n, 7)$ and $Add(BF^n, 8)$. Assume that $Query(BF, 20)$ is positive due to the Bloom filter's false positive, so (Step 4) $Set_{diff} = \{(7,1), (8,1), (20,3)\}$. Because keys 7 and 8 are sequential in the sorted Set_{diff} and share the same `tid`, as in Figure 3(b), they can be associated with point 8 alone (Step 5).

Turning back to Step 2, in the meantime, Set_{fly}^{n} collects 4-tuples and keys for new coming connections, e.g, it sets $Set_{fly}^{n} = \{9, 19\}$. Assume that $Query(BF^n, 9)$ and $Query(BF^n, 19)$ are both false positives. We use the eq. 1 and hashes in Figure 3 to shrink Set_{fly}^{n} (Step 6):

1) $H_{now}^{n}(9) = 2$ and $H_{diff}^{n}(9) = 3$, so retain 9.
2) $H_{now}^{n}(19) = H_{diff}^{n}(19) = 3$, so remove 19.

Thus $Set_{fly}^{n} = \{9\}$, and the adjustment is done.

The key insight here is based on the observation that although the TCP processing threads have a per-connection status, namely, the TCP connection table, these are inadequate for the packet I/O thread to query in real time for severe cache misses and lock contention. Another advantage of the CSB is that the mapping of a key to a `tid` is determined by finding points on the unit circle, most numerical neighboring keys are related to the same `tid`, and streams will be closed as time elapses. Therefore, the CSB can construct a compact H_{diff} to record massive key-`tid` mappings in a compact manner.

For an active open connection from thread `tid`, the source port is selected by testing the 4-tuple with $tid = CSB(4 - tuple)$. This is intended to allow the application to give up its freedom to use a specific port in exchange for connection locality.

IV. Implementation Details

Janus is in full compliance with standards RFC 793 and RFC 1122 and supports TCP options (RFC 7323), as well as time stamps, selective ACKs, and congestion control. We tested its conformance to the RFCs using `packetdrill` [6], which is an open-source scripting tool that enables testing the correctness and performance of entire network stack implementations.

In Janus, each connection has a 544-byte TCP Control Block (TCB). Running an application in the same thread with the TCP stack can greatly reduce the per-connection memory cost. The benefits are twofold. On the one hand, Janus uses a per-thread buffer pool (huge pages that are organized by `rte_ring`) to receive data from the NIC, so that per-connection receive buffers are unnecessary. On the other hand, each thread has only one active coroutine at any given time, so there is no need for the idle connection to reserve a send buffer. As a result, each idle connection only costs around 600 bytes, so the total consumption for 40 million connections is 24GB.

The CSB's consistent hash is implemented with a sorted array of (point, bucket) pairs. The bucket corresponding to a given key value is located through a binary search. The key values are truncated to 32 bits to save space. Unlike the classic Red-Black Tree implementation, the sorted array does not support dynamic updates efficiently: in order to change the number of buckets, the entire data structure must be rebuilt. But it is more compact (8 bytes per point per bucket), and rebuilding is not a big deal, for it is processed in the control plane. The CSB's Bloom filter is implemented with the Murmur hash function [2].

DPDK's `rte_mbuff`, which is used to is also modified to store per-packet metadata such as a SACK flag and resend times. The metadata reside in the same continuous memory as the packet descriptor, which Janus can access efficiently.

V. Evaluation

In this section, we first present micro-benchmarks for the CSB and coroutines. Then we evaluate Janus's performance and compare it with CentOS 7.4[4] (kernel 3.10.0) and mTCP[5] v2.0.

All benchmarks were conducted on two Dell PowerEdge R730 servers, each of which has two Intel Xeon CPUs, E5-2698 v3 @ 2.3-3.6GHz (16 cores, hyper-thread ON), 128GB DDR3 2133Mhz memory, and two Intel 82599ES dual-port 10Gb Ethernet adapters (4x10GE). The servers run CentOS 7.4 (kernel 3.10.0).

A. Micro-Benchmarks

This section presents the micro-benchmark results for our CSB mechanism and coroutine implementations.

1) CSB Benchmark: We present the CSB's adjustment time and lookup speed for different numbers of concurrent streams. More concurrent streams will cost more adjustment time for building H_{diff} and BF. The CSB's Bloom filter was set with a false positive rate of 0.0001. For the consistent hash algorithm, more points per bucket will have a smaller deviation but induce a greater time complexity. We measure the deviation by computing the standard error of the fraction of hash values assigned to each bucket. To simulate massive streams, we generate streams with randomly generated 4-tuples and assign 32 packets to each stream. Then we generate a hash key for each 4-tuple with RSS. Table III shows the evaluation results for the key lookup and adjustment times. The CSB is set to 16 buckets (TCP processing threads), 16 points per bucket, and uses a 1-packet I/O core.

2) Coroutine Benchmark: We wanted to measure the relative overhead of coroutines compared to function calls (callbacks). First, we needed to determine the C language function call overhead as a baseline. Using a 1KB stack with random data, the overhead was constant in the recursion depth and took 32 ns on our server. Then we tested the yield and resume costs with various numbers of concurrent coroutines. With a 1KB stack size, the overhead of our coroutine implementation was about twice as much as for the C function call, and it was nearly invariable for the number of concurrent coroutines. We believe that such overhead could be amortized by batching at the application level.

TABLE III: Performance of CSB for different numbers of concurrent streams

Concurrent Streams	**Lookup Speed**	**Adjustment Time (ms)**	**Standard Error**
1,000,000	14.4M key/sec	180	5%
4,000,000	12.8M key/sec	400	5%
10,000,000	11.2M key/sec	1281	6%
20,000,000	9.8M key/sec	2399	5%

B. Experimental Setup

We implemented three similar simple HTTP servers with Janus API, mTCP API, and socket API to compare their performance. The HTTP server receives an HTTP request and responds with a variable-sized message.

We used an IXIA traffic generator [13] to initiate massive HTTP requests, and the server responses with a fixed message. The Maximum Transmission Unit (MTU) was set to 1500 bytes. To simulate a real production environment, different from the experiment setup in [14], established connections were gracefully terminated with the `FIN` flag set to 1. For Linux, we changed the kernel's default parameters to unleash its performance, e.g., we stopped `firewalld`, shrank the TCP buffer size, increased `fs.nr_open`[6], and so on. In all test cases, Janus used 8 CPU cores (for Janus, 1 packet I/O and 7 TCP processing cores) and 64-byte messages per connection, unless otherwise specified.

C. Scalability on Multicore Systems

Figure 4 shows that Janus scales almost linearly with the number of CPU cores. Since mTCP's number of threads is limited by the RSS queues, we needed to modify its I/O engine to support more cores. Without loss of generality, the comparison ends with 8 CPU cores.

Janus's connection per second (cps) rate is lower than that of the other approaches for 2 cores, for there is only 1 core processing TCP and 1 doing packet I/O. However, when more cores are added, Janus outperforms the other approaches.

Then we added more cores to explore Janus's scalability limitations. We can see in Figure 5 that Janus scales linearly until 14 cores are reached. Then the cache miss rate reaches nearly 50%, which causes performance degeneration. The share-nothing architecture was able to avoid

[4]CentOS 7.4 supports `SO_REUSEPORT` and was the latest CentOS release at the time of experiment.

[5]mTCP is currently one of the most advanced user-level stacks.

[6]The maximum number of socket descriptors per process, which we set to 20,000,500.

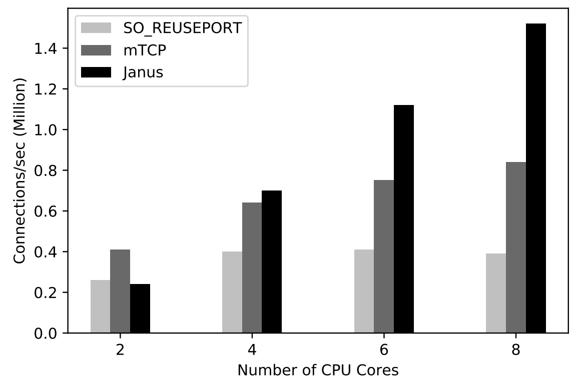

Fig. 4: Comparison of connection accept throughputs.

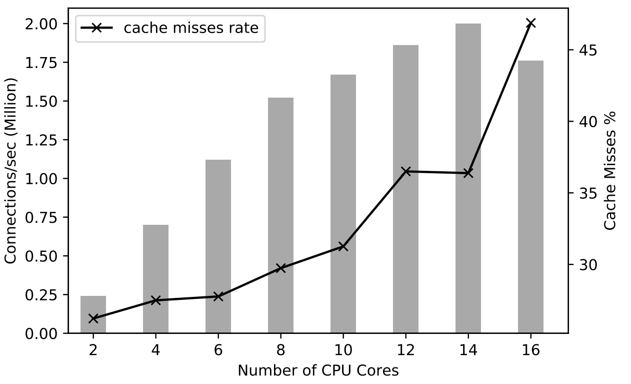

Fig. 5: Scale to 16 CPU cores.

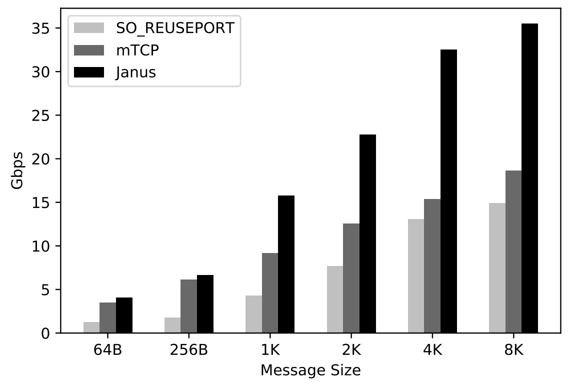

Fig. 6: Bandwidth for different message sizes; all approaches use 8 cores.

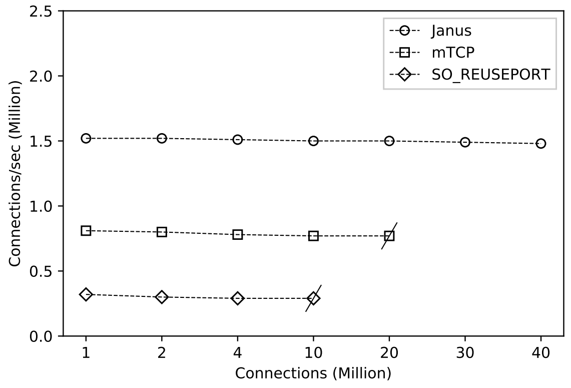

Fig. 7: The connection establishment speed varies with the number of concurrent connections.

data structure access contentions. However, the ultimate ceiling for multicore scalability was determined by the shared L3 cache.

D. Throughput

Figure 6 shows the throughput for varying message sizes. The connection is closed immediately after the message transmission. Janus's performance improvement is noticeable for all message sizes. All approaches were able to achieve 10 Gbps for a 4KB message size. Janus was able to reach 35.49Gbps for 8KB message sizes, which is twice that of the other approaches. When augmented to 10 cores, Janus could saturate a 40Gbps link with an 8KB message size.

E. Concurrency

To several protocols use long connection, such as HTTP/2 and WebSocket, Janus was optimized for massive concurrency. We initiated a specific number of connections that did not close so that we could evaluate the concurrency with the connection establishment speed.

Figure 7 shows the cps for different numbers of concurrent connections; all approaches were able to maintain a stable acceptance speed. However, the test of Linux was aborted at 10 million concurrent connections, for the OS became stuck and unable to proceed any further. mTCP run out of memory at 20 million concurrent connections. Janus was still able to maintain a reasonably high cps (1.48 million) with 40 million concurrent connections. This can be attributed to the compact data structure and share-nothing architecture. In other words, partitioning the TCP connection table enables threads to look up packets in parallel.

F. Latency

Traditionally, there is a trade-off between throughput and latency. In order to achieve higher throughput, Janus processes packets in batches, which is made possible with the DPDK PMD. In addition, the pipeline between packet I/O threads and TCP processing threads could introduce further latency. We measured the three-way handshake latency [7], as shown in Figure 8. Under a workload of 0.8 million cps, both mTCP and Linux endured a severe packet loss, so beyond that speed, only the results for

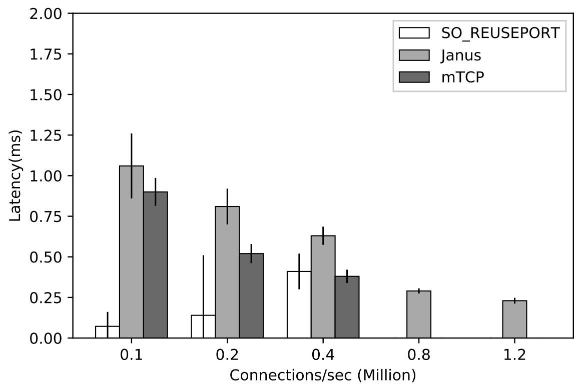

Fig. 8: Three-way handshake latency.

Janus are shown. At 0.1 million cps, Janus's connection latency was much higher than Linux's and mTCP's. Owing to its batch processing, Janus' latency is lower when more connections are accepted, and it achieves a latency of 0.23 milliseconds at 1.2 million cps. We believe that one possible optimization in this case would be to adjust the batch sizes dynamically [19].

VI. Conclusion

In this paper, we have proposed a highly scalable TCP stack to serve massive numbers of concurrent TCP connections. In addition to existing optimization techniques, Janus separates I/O cores from processing cores and lets applications run as per-connection coroutines in the packet processing threads. This design choice greatly improves data locality, memory efficiency, and programming flexibility. We have also proposed a novel TCP load balancing algorithm, the Consistent Stream Balancer, which combines a Bloom filter and consistent hashing to allow live reconfiguration of the load-balancing hash function if the flow distribution to the cores is unbalanced. Our evaluations show that Janus can accept 1.86 million connections per second while maintaining 40 million concurrent connections. Moreover, Janus outperforms Linux 3.10.0 and mTCP by 3.9 and 1.8 times, respectively.

VII. Acknowledgement

The authors would like to thank the anonymous reviewers for their valuable comments and helpful suggestions. This work is supported by National Key R&D Program of China (Grant NO. 2016YFB0801300).

References

[1] Data plane development kit. http://www.dpdk.org/, 2014.

[2] A. Appleby. Murmurhash 2.0, 2008.

[3] 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 the 11th USENIX Symposium on Operating System Design and Implementation (OSDI)*, number EPFL-CONF-201671. USENIX, 2014.

[4] M. Belshe, M. Thomson, and R. Peon. Hypertext transfer protocol version 2 (http/2). 2015.

[5] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[6] N. Cardwell, Y. Cheng, L. Brakmo, M. Mathis, B. Raghavan, N. Dukkipati, H.-k. J. Chu, A. Terzis, and T. Herbert. packetdrill: Scriptable network stack testing, from sockets to packets. In *USENIX Annual Technical Conference*, pages 213–218, 2013.

[7] N. Cardwell, S. Savage, and T. Anderson. Modeling tcp latency. *Proceedings - IEEE INFOCOM*, 3:1742 – 1751, 2000.

[8] M. Corp. Receive side scaling. http://msdn.microsoft.com/library/windows/hardware/ff556942.aspx, 2014.

[9] I. Fette. The websocket protocol. 2011.

[10] R. Graham". "the secret to 10 million concurrent connections-the kernel is the problem, not the solution". 2013.

[11] S. Han, S. Marshall, B.-G. Chun, and S. Ratnasamy. Megapipe: A new programming interface for scalable network i/o. In *OSDI*, pages 135–148, 2012.

[12] M. Honda, F. Huici, C. Raiciu, J. Araujo, and L. Rizzo. Rekindling network protocol innovation with user-level stacks. *ACM SIGCOMM Computer Communication Review*, 44(2):52–58, 2014.

[13] IXIA. Ixia breakingpoint. https://www.ixiacom.com/products/breakingpoint, 2013.

[14] E. Jeong, S. Woo, M. A. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park. mtcp: a highly scalable user-level tcp stack for multicore systems. In *NSDI*, pages 489–502, 2014.

[15] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pages 654–663. ACM, 1997.

[16] M. Kerrisk. The so_reuseport socket option. https://lwn.net/Articles/542629/, 2013.

[17] X. Lin, Y. Chen, X. Li, J. Mao, J. He, W. Xu, and Y. Shi. Scalable kernel tcp design and implementation for short-lived connections. In *ACM SIGPLAN Notices*, volume 51, pages 339–352. ACM, 2016.

[18] I. Marinos, R. N. Watson, and M. Handley. Network stack specialization for performance. In *ACM SIGCOMM Computer Communication Review*, volume 44, pages 175–186. ACM, 2014.

[19] J. C. McCullough, J. Dunagan, A. Wolman, and A. C. Snoeren. Stout: An adaptive interface to scalable cloud storage. In *Proc. of the USENIX Annual Technical Conference–ATC*, pages 47–60, 2010.

[20] MigratoryData. Migratorydata server. http://migratorydata.com/, 2016.

[21] L. NAPI. https://wiki.linuxfoundation.org/networking/napi, 2016.

[22] ntop. Pf_ring zero copy. http://www.ntop.org/products/packet-capture/pf_ring-zc-zero-copy/, 2015.

[23] A. Pesterev, J. Strauss, N. Zeldovich, and R. T. Morris. Improving network connection locality on multicore systems. In *Proceedings of the 7th ACM european conference on Computer Systems*, pages 337–350. ACM, 2012.

[24] S. Peter, J. Li, I. Zhang, D. R. Ports, D. Woos, A. Krishnamurthy, T. Anderson, and T. Roscoe. Arrakis: The operating system is the control plane. *ACM Transactions on Computer Systems (TOCS)*, 33(4):11, 2016.

[25] R. Reed. Scaling to millions of simultaneous connections. *Erlang Factory SF*, 2012.

[26] L. Rizzo. Netmap: a novel framework for fast packet i/o. In *21st USENIX Security Symposium (USENIX Security 12)*, pages 101–112, 2012.

[27] J. R. Von Behren, J. Condit, and E. A. Brewer. Why events are a bad idea (for high-concurrency servers). In *HotOS*, pages 19–24, 2003.

[28] K. Yasukata, M. Honda, D. Santry, and L. Eggert. Stackmap: Low-latency networking with the os stack and dedicated nics. In *2016 USENIX Annual Technical Conference (USENIX ATC 16), Denver, CO*, 2016.