Improving Connection Scalability in TAS

Rajath Shashidhara, Piyush Kumar Jain, Sundara Raman Ramachandran
Department of Computer Science, The University of Texas at Austin
{rs56566, pm29839, sr47262}

Abstract

TCP Acceleration as a Service (TAS) is a DPDK based userspace TCP stack implementation particularly designed to optimize RPCs in the datacenter. In comparison to Linux and other kernel bypass TCP implementations, TAS is shown to achieve significantly higher throughput and lower latency. In this report, we perform experiments to stress test TAS with very large number of connections to identify the bottlenecks. On analysis, we find that the overhead of Queue Manager – Fair Queueing (FP) based flow scheduler in TAS increases with the number of connections. In place of it, we propose the use of Carousel – a scalable traffic pacing algorithm to replace the existing flow scheduler to improve connection scalability of TAS. Our experiments show that Carousel conforms to rate limits, achieves ~1.2x times higher throughput and decreases CPU utilization of enqueue operation by 50% in comparison with FQ pacing for connections greater than 16k. In addition, we analyze the performance of short-lived connections for RPC workloads. TAS incurs overheads on connection setup and teardown due to the fast path and slow path split design. We present solutions to alleviate this problem and evaluate them against the existing approach.

1. Background

TAS fast path is heavily optimized to handle high throughput TCP traffic. Excellent performance is observed even for 1000s of connections. TAS maintains limited state per flow in memory (~128 bytes per flow) and the flows are hashed into a table. Minimal flow state implies that a large number of flow states can be stored in cache. On packet arrival, the corresponding flow state is efficiently looked up from the hash table in constant time $O(1)$.

In addition, the packet scheduler in TAS is implemented using the FQ (Fair Queuing) pacing technique. Flows are kept sorted based on the time of the next transmission – computed using the congestion control rate, buffer occupancy and past average throughput. The sorted list is implemented as a skip list — a randomized data structure that closely approximates the on average performance of a balanced binary tree. While skip list performs very well in practice, it is the only component in fast path for which the overhead grows as $O(log\ N)$ as the number of connections N as becomes very large. Queue manager constitutes a major portion of the total CPU cycles spent by the fast path thread. As seen in Figure 1, fast path spends ~18% of the total CPU cycles. Moreover, randomized nature of skip list introduces random jumps in the access patterns and makes it harder to prefetch elements into the cache. Results in Table 1 confirm this theoretical argument — the average number of cycles spent by the fast path thread in performing a single enqueue operation in a skip list data structure increases with the increase in number of flows. This presents us with an opportunity for improvement.

Pacing is a crucial requirement in network stacks as it helps to limit flows to transmit within a target rate set by the congestion control algorithm in the stack. Exceeding the target rate set can cause packet drops and also affects fairness. Pacing also helps maintain a check on the burstiness of flows which is also a likely cause for packet drops due to shallow switch buffers. To pace packets for a flow while achieving a specified target rate, networks stacks transmit packets at uniform intervals. In essence, inter-packet time gaps

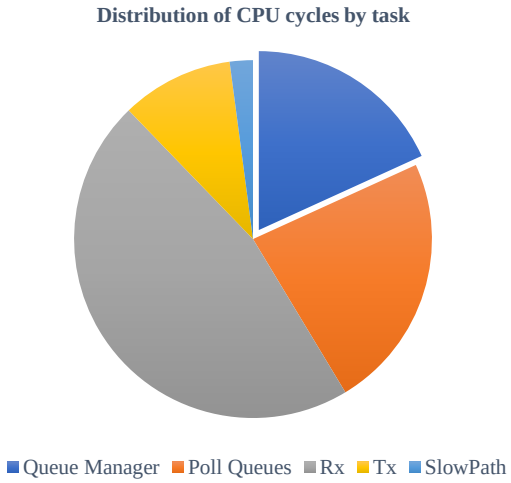

Figure 1: Distribution of CPU cycles by task in TAS

(computed based on the target rate) are inserted between consecutive packets that are to be sent.

To achieve pacing, one requires to maintain a data structure which maintains an ordering of flows based on time stamps and exposes two abstract methods – *enqueue(flow, time_stamp)* and *dequeue()*.

Timestamps are needed to estimate when next a flow is allowed to transmit based on its target rate. An ordering is needed to determine which flow is the next eligible candidate permitted to transmit.

TAS uses a skip-list data structure for maintaining an ordering (a total ordering in this case). It does so by assigning timestamps to flows (let's call it *next_ts*, represents when the data from a flow needs to be transmitted next) as soon as packets are transmitted from that flow. *next_ts* is computed based on the rate set for the flow by the congestion control algorithm in the slow path, the maximum segment size of the flow and the time of last transmission. A skip-list allows to perform *dequeue* operations in $O(1)$ time, and *enqueue* in $O(log\ \text{num\_entities})$ time.

To pace packets appropriately, the fast path thread ***periodically polls*** the skip list for flows which require data transmission (i.e., if *next_ts* < *curr_ts*, where curr_ts is the current timestamp in the fast path thread). Polling for flows involves removing the flow from the head of the skip list (an inexpensive *dequeue* operation), followed by enqueuing the flow back in the skip list (in case more data is

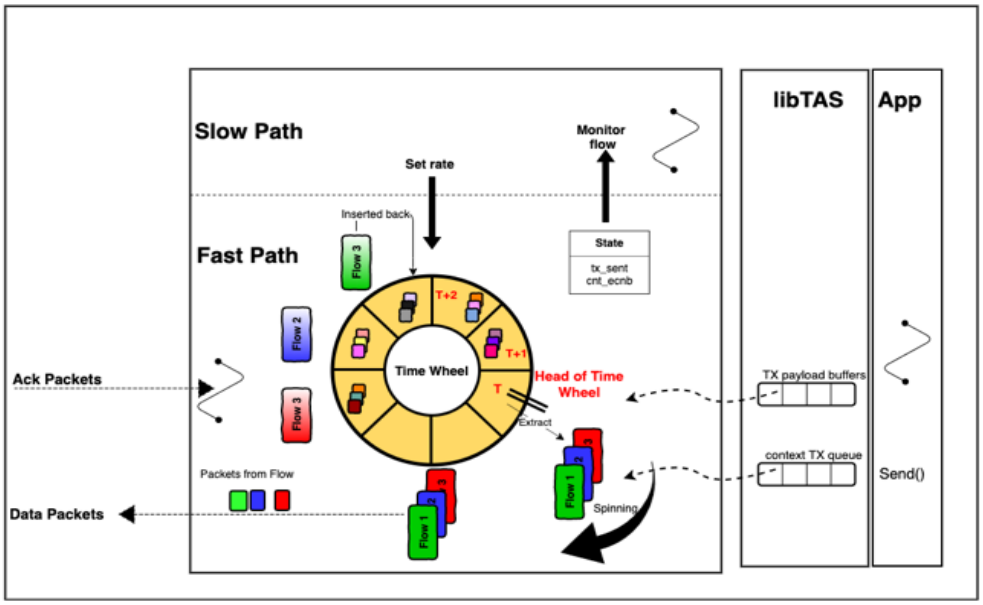

Figure 2: An overview of TAS with Carousel depicting how flows are enqueued and dequeued in and out of the timewheel.

available in the application's TX buffer, which is the common case).

Enqueuing the flow back in the skip list is an expensive operation as it involves traversing the skip list for insertion. In case the application TX buffer for the flow is empty, it is not enqueued back into the skip list, but *next_ts* is still updated. The flow is later enqueued in the skip list by the fast path thread when the application issues a TX bump.

The hypothesis that *enqueue* in TAS's skip-list becomes slower with increasing number of connections was confirmed by the following experiment (Table 1) - An echoserver benchmark with server running on 20 cores served by TAS limited to run on a single core. The client was run with 64 cores with the number of connections per core ranging from 64 to 1536. A message size of 64 bytes was used and each echo benchmark was run for 120 seconds. For each run, all connections were opened before transmission of the first packet from the client. The DCTCP congestion control algorithm was enforced. Also, the client was configured to place at most 500 messages in the pipeline (i.e., not echoed back by the server).

2. Solution & Implementation

To solve this problem, we can relax the condition of maintaining a total order of the flows on the next_ts field without compromising the pacing. "Carousel: Scalable Traffic Shaping at End Hosts" (A. Saeed et al, SIGCOMM'17) proposes the use of a timewheel to pace packets in the network stack by maintaining partial ordering of the flows. Our solution adopts the same technique to replace the skip list by a timewheel, which supports constant overhead *enqueue* operation irrespective of the number of flows managed by the queue manager. The *dequeue* operation will remain an amortized O(1) time complexity operation. The tradeoff is between accuracy of rate conformance (depends on the level of ordering) and performance of the enqueue operation. Coarser ordering will allow for cheaper *enqueue* operations but hurt rate conformance. A point to note here is - for a busy polling system such as TAS, there is a maximum bound on the level of ordering that can be maintained. Since the fast path thread performs various tasks between two successive batches of *dequeue* operations (say the turn-around time to return to *dequeuing* tasks is *t* seconds), there is no point in maintaining ordering between the flows which have their *next_ts* fields within *t seconds apart*, as they will all be dequeued at the same time anyway. We exploit this maximum bound on ordering to make the *enqueue* operation less expensive while not hurting rate conformance when compared to a total ordering.

Algorithm 1 Algorithm for Insertion into Timewheel

```
1: virtual_Ts                                              ▷ Current Time stamp
2: procedure FLOW_ACTIVATE_TIMEWHEEL(flow)
3:     Ts_rate ← GET_NEXT_TS_BASED_ON_RATE(flow)
4:     Ts ← MIN(flow.next_Ts, Ts_rate)
5:     Ts ← MAX(Ts, virtual_Ts)
6:     delta ← Ts − virtual_Ts
7:     pos ← delta/Granularity
8:     timewheel_idx ← WRAP_AROUND(pos)
9:     ADD_FLOW_TO_TIMEWHEEL_BUCKET(flow, timewheel_idx)
```

Algorithm 2 Algorithm for Extraction from Timewheel

```
1: timewheel_head_idx                                      ▷ Head index in Time wheel
2: procedure POLL_TIMEWHEEL(num_flows, cur_Ts)
3:     timewheel_idx = timewheel_head_idx
9:         timewheel_idx += 1
10:        num_buckets_visited += 1
11:    end while
12:    timewheel_head_idx ← timewheel_idx
13:    virtual_Ts += Granularity * num_buckets_visited
14:    for each flow in flows_extracted do
15:        flow.next_Ts ← GET_NEXT_TS_BASED_ON_RATE(flow)
16:        if flow.avail_data > MSS then
17:            FLOW_ACTIVATE_TIMEWHEEL(flow)
18:        end if
```

* A "hop" here refers to adding a message in a shared queue between two components. TAS) will fetch the message from the queue periodically while performing other tasks a

The implementation of Carousel presented in [2] enqueues references to packets in the timewheel. Our implementation differs from this in two aspects –

1. Flows ids are enqueued in the timewheel instead of packet references. This is possible as shaping is done before packets are sent to the driver as compared to the implementation in [2], where the timewheel paces in the NIC (outside of the protocol stack). The fast path has access to the transmission buffers for all flows and doesn't need packet references.
2. The carousel implementation in [2] uses linux TCP network stack which has TSQ that maintains two packets outstanding between the stack and the NIC (and hence in the timewheel). Since TAS manages both the network stack and the pacing, this is not required.

Pacing using a timewheel consists of three steps for each flow –

1. Computation of *next_ts* for the flow.
2. Enqueueing the flow in the timewheel
3. Dequeuing the flow from buckets in the timewheel which have flows with timestamps earlier than the current time stamp. Note that a flow is enqueued back into the timewheel immediately after a dequeue in case it has more data available to be transmitted.

The timewheel is implemented in TAS as a circular array of buckets, each of which represents a particular timestamp and can hold any number of flows. Buckets have a certain granularity associated with them, i.e., a bucket with timestamp *t* in a timewheel of granularity *g* will hold flows that have a *next_ts* in the range $t -> t + g$. The maximum *next_ts* that a timewheel can support is $g * number\ of\ buckets$. This corresponds to the minimum rate which the timewheel can conform to.

3. Evaluation - Connection scalability

Testbed cluster: Our evaluation cluster contains a 72-core (144 hyper threads) Intel Xeon Gold 6154 system at 3 GHz with 187 GB RAM, 27 MB aggregate cache, and an Intel Corporation 82599ES 10 Gb Ethernet adapter. We use this system as the server and the client. We run Debian 10 with DCTCP congestion control on all machines. We use a Netberg Aurora 720 100G Ethernet switch.

Baseline: We compare the performance of timewheel based pacing to the skip-list based FQ pacing. The experiment is performed on the pipelined RPC echo benchmark on a 20-core server. A granularity of 1 µs and a timewheel length of 5 * 10^5 µs was used for the experiments. This was chosen as the fast path thread turn-around time between two batches of dequeues was about 1 µs. A message size of 64Kb was used for performing the tests.

Figure 3 shows the average throughput achieved by both variants and it can be inferred that pacing using a timewheel performs better. The number of drops and retransmissions for Carousel and FQ pacing are comparable which ensures that Carousel is as rate conformant as FQ pacing.

We observe a 10% increase in throughput for 90k connections and 20% increase in throughput for 32k connections with carousel compared to FQ pacing. Table 1 shows the average cycles spent by the fast path thread to perform one *enqueue* operation. The average cycles per enqueue at lower number of total connections show that the fixed cost of enqueueing in a timewheel is lower as compared to a skip-list (this can be attributed to the simplicity of the enqueue operation in the timewheel). We can also infer that with increasing number of flows, the cost of enqueue in a timewheel stays fairly constant, while it increases more than 3-fold in a skip-list.

Below 16k connections we see that TAS with timewheel performs poorly as compared to FQ pacing. A possible explanation is - for less number of connections, buckets in the timewheel are sparsely occupied and thus we have to pay the cost of checking each bucket before we get to the bucket with appropriate flows for transmission.

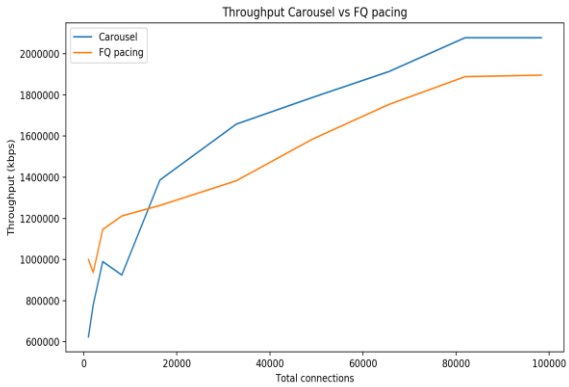

Figure 3: Throughput comparison of TAS with Carousel & TAS with FQ pacing for pipelined RPC echo benchmark on 20 core server.

Total Connections	Carousel	FQ pacing
1024	69.70	120.48
2048	70.13	125.23
4096	70.89	131.04
8192	73.57	142.29
16384	77.15	160.40
32768	80.79	169.77
49152	84.47	218.08
65536	92.02	359.80
83200	93.48	397.01
98304	99.85	395.42

4. Analysis of short-lived RPC workloads

Apart from improving the queue manager in TAS for large number of flows, another scenario is to handle short-lived connections. TAS is optimized for handling common case TCP operations and not for rarely occurring tasks and exceptions. These are handled by the slow path to avoid instruction cache pollution and high amount of branching in the fast path thread. For the case of short-lived connections, connection setup and tear-down (high overhead

Table 1 Average number of cycles spent in enqueuing a flow for varying number of flows handled by the queue manager.

The TAS paper used an echoserver benchmark with a fixed number of total active connections with varying number of messages served per connection to simulate short-lived behavior. On any connection close, a new one was immediately created to keep the total number of connections constant. Results show that for short-lived connections with more than 4 messages sent per connection, TAS achieves higher throughput in comparison to Linux. However,

below that limit, performance of TAS is lower than Linux and we analyze this workload to identify the bottlenecks and propose solutions.

Number of hops (*) analysis accepting a connection – For a server running on TAS, all SYN packets (new connection requests) are forwarded by the fast path to be handled by the slow path (1st hop). The slow path adds these to the backlog queue it maintains for each application. It then issues a notification (EPOLLIN) to the application on its listen file descriptor (2nd hop). Following that, the application issues an accept() call - creates a file descriptor and adds a message in the app to slow path shared memory space (3rd hop) and return immediately (assume non-blocking accept). The slow path on receiving the accept performs initialization tasks for congestion control, structures in the fast path, shared memory space between fast path and application, and registration of connection in the application data structures in the slow path. Post this, the slow path asynchronously performs two operations – sending the SYN_ACK and notifying the client of the completed accept by issuing an EPOLLIN on the listening file descriptor (4th hop*). The client, on receiving this, again calls an accept() to find an already created connection (it doesn't create a file descriptor this time).

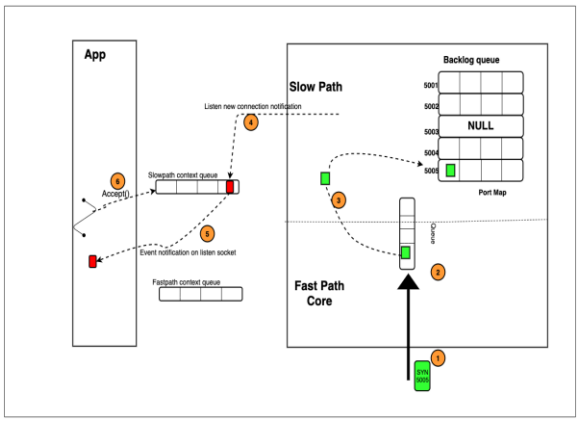

Figure 4: Connection setup workflow in TAS

Figure 4 depicts the process till the first accept is issued by the application (3 hops – fast to slow, slow to app, and app to slow). The 4th hop depends on whether the accept is blocking or not. A possible optimization to reduce the overall waiting overhead is discussed below.

We identified four bottlenecks in the connection setup/tear-down –

1. **Connection lookup using hash table:** The slow path thread maintains a linked list of connection holding data structures for each application. On receiving a request to close a connection, the slow path thread performs an expensive linear search through the linked list to find the connection to be closed. We added a per application hash table data structure to lookup connections based on the 4 tuple (src ip, src port, rmt ip, rmt port) which helped in avoiding this cost.

2. **Removal of connection from list:** Open connections are maintained in a per application context singly linked list. On connection close, it has to be removed. This requires traversing through the entire linked list. We modified the linked list to a doubly linked list to convert the O(n) time overhead connection removal operation into a constant overhead one.

3. **Removal of connection from list in Congestion Control:** This is similar to the optimization in 2, on the linked list of connection structures that store congestion control information.

4. Each *accept* requires at least three hops to result in a successful connection. Each hop is dependent on the previous as depicted in Figure 4 and hence they can only occur one after the other. However, the first and the second hop can be decoupled – the fast path can simultaneously notify both the application and the slow path about a possible new connection request. The first hop stays the same where the fast path notifies the slow path of the SYN request. The fast path can also notify the corresponding application based on the port map (this will result in the EPOLLIN on the listen socket). This will reduce the fixed total time required to accept a new connection (Figure 5). Since connection setup forms a majority part of the benchmark for short-lived connections, creating connections faster will result in less overall time spent in connection creation during the run and hence more connections will be opened and closed in a single run. This also implies an increase in throughput, given the constant number of messages sent per connection created. Also, as multiple fast path cores handle connection requests, moving the task of notifying the application to the fast path helps in parallelizing the notification task and not restricting it to a single slow path core.

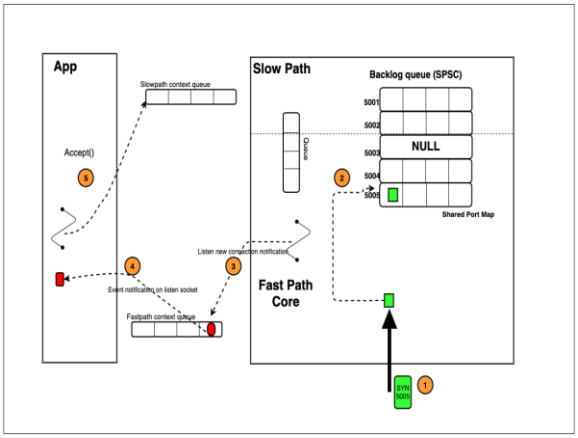

Figure 5: Modified connection setup workflow to reduce the number of hops required to notify the application of new incoming connections.

5. Evaluation – Short lived connections

Testbed cluster: Our evaluation cluster contains a 4-core (4 hyper threads) Intel Xeon CPU E3-1220 v5 system at 3 GHz with 7.8 GB RAM, 8.3 MB aggregate cache, and an Intel Corporation 82599ES 10 GB Ethernet adapter. We use this system as the server and the client. We run Debian 10 with DCTCP congestion control on all machines. We use a Netberg Aurora 720 100G Ethernet switch.

	Opt1	Opt2	Opt3
With optimization	427	127	77
Without optimization	6968	3500	2714

Table 2: Comparison between average number of cycles spent in the respective function corresponding to each optimization. These functions constitute a small portion of the total runtime in the slow path and hence don't result in high performance gains (in terms of throughput).

The cycles reduced by each optimization per connection remove are shown in Table 2. It was expected that the fourth optimization (which would reduce the turnaround time for an accept from 3 to 2 hops) results in a throughput improvement, but we found that the throughput remained the same.

6. Conclusion

We present a version of TAS with Carousel for improving the connection scalability of vanilla TAS. We show that the proposed version of TAS with carousel performs better(~1.2x) compared to the existing TAS implementation for connections greater than 16k. We also provide few performance optimizations for improving the performance for short-lived connections and evaluate their effect for short lived RPC workloads.

7. Future Work

1. Further analysis of the fourth optimization for short lived connections.
2. Pre-warm connections in the slow path to improve performance for short-lived connections, inspired from [3]
3. Circular FFS-based Queues and approximate queuing presented by [4] should be experimented with TAS.
4. Analyzing the performance of timewheel in TAS in conjunction with TIMELY congestion control.

8. Code

Carousel implementation - https://github.com/pkj415/tas/tree/carousel-testing-piyush-without-debt

Short lived optimizations - https://github.com/pkj415/tas/tree/short-lived-optimizations

Acknowledgements

We thank Prof. Simon Peter and teaching assistant Tim Stamler for their helpful discussions and constructive feedback.

References

[1] Antoine Kaufmann, Tim Stamler, Simon Peter, Naveen Kr. Sharma, Arvind Krishnamurthy, and Thomas Anderson. 2019. TAS: TCP Acceleration as an OS Service. In *Fourteenth EuroSys Conference 2019 (EuroSys '19), March 25–28, 2019, Dresden, Germany*. ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3302424.3303985

[2] Ahmed Saeed, Nandita Dukkipati, Vytautas Valancius, Vinh_e Lam, Carlo Contavalli, and Amin Vahdat. 2017. *Carousel: Scalable Traffic Shaping at End Hosts. In Proceedings of* SIGCOMM '17, Los Angeles, CA, USA, August 21-25,2017, 14 pages. DOI: 10.1145/3098822.3098852

[3] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. 2017. *My VM is Lighter (and Safer) than your Container*. In Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17). ACM, New York, NY, USA, 218-233. DOI:https://doi.org/10.1145/3132747.3132763

[4] Ahmed Saeed, Yimeng Zhao, Nandita Dukkipati, Ellen Zegura, Mostafa Ammar, Khaled Harras, Amin Vahdat. 2019. *Eiffel: Efficient and Flexible Software Packet Scheduling.* 16th USENIX Symposium on Networked Systems Design and Implementation NSDI'19. Boston, MA. 978-1-931971-49-2. https://www.usenix.org/conference/nsdi19/presentation/saeed