Implementation and end-to-end Evaluation of the HULL Architecture

Semester Project
Prasopoulos Konstantinos
School of Computer & Communication Sciences, EPFL, Switzerland
Data Center Systems Laboratory, Prof. Edouard Bugnion
Supervised by Mia Primorac

***Abstract*—User-facing data center applications, such as Web search, online store etc., have very strict tail latency requirements in order to offer a satisfying experience to all users. Because of the large amount of queried data that need to be stored in a distributed fashion, the communication patterns of these applications often involve fanning-out queries to multiple servers and aggregating the responses. The final response time is determined by the latency of the slowest response and thus, latency variability needs to be bounded. One important source of latency variability is in-network queuing delay that needs to be minimized. One of the proposals to reduce queuing-induced latency is the HULL architecture. The goal of this project is to implement the HULL architecture and evaluate it on workloads that model such applications via simulation. HULL aims for near baseline fabric latency along with acceptable bandwidth utilization by employing three techniques: congestion signals calculated based on part of the link capacity to avoid queue buildup, packet pacing for throughput-heavy flows to avoid queuing spikes and DCTCP in order to respond to congestion proportionally and avoid throughput loss. On tested workloads with tail-latency requirements in the order of tens of milliseconds, HULL performs no better than DCTCP and up to 40% better than TCP Reno, depending on the background load. On the other hand, on workloads with tighter tail-latency requirements (hundreds of microseconds - e.g. an in-memory database) and deterministic server processing times, HULL reduces query completion latencies by almost 50% compared to DCTCP. However, this advantage is reduced to 11% when the server processing times vary.**

I. Introduction

The way users experience the Internet, be it interacting on social media, querying a search engine etc., is highly dependant on the fast response times of Online Data Intensive (OLDI) applications. Upon arriving to a data center, such application requests often follow the fan-out/fan-in pattern in which the request is multiplied and assigned to many servers that contain different shards of data. The responses are then aggregated to produce the final page content. Satisfying the request either means that all the servers must respond or it involves returning a partial, lower quality result. This decision is made based on response deadlines called *Service-Level Objectives* (SLOs). Emphasis is given on the 99^{th} percentile of overall response times in order to provide a consistent experience for the vast majority of users. The fact that satisfying the overall request depends on the slowest server means that it is very hard to respect tight SLOs, especially as the number of servers increases (The Tail at Scale problem [1]).

OLDI applications share the data center infrastructure with large, throughput sensitive background flows that synchronize the state of the servers or perform batch jobs. TCP can provide good throughput for the background flows but, because it detects congestion only after network buffers are full, it significantly reduces the performance of latency-sensitive flows. A full switch buffer can add several milliseconds of latency which could be used more productively by the algorithms of a search engine or could be subtracted from the SLO. It is thus important to operate with limited buffering within the network. At the same time, some amount of queuing is necessary to maintain acceptable throughput levels.

A different kind of congestion control that combines low network-induced latency through minimal queuing and achieves acceptable throughput is necessary. The HULL (High-bandwidth Ultra-Low Latency) architecture [2] attempts this by employing three mechanisms. First, it uses *Phantom Queues* (PQs). PQs are positioned on a link and behave like queues that drain at a lower rate than the link's capacity (without actually queuing packets). In this way, this virtual queue will build-up and signal congestion earlier than a switch buffer draining at the nominal rate. Therefore, the PQ attempts to keep the aggregate rate of the senders bellow that which the switch can handle in order to achieve minimal buffering. PQs use ECN (Explicit Congestion Notification) marking in conjunction with the second mechanism, DCTCP (Data Center TCP) [3]. In contrast to TCP which reacts to the presence of congestion by halving the congestion window, DCTCP reacts to the extent of congestion (estimated by the number of ECN-marked packets) in a proportional manner. DCTCP also utilizes the active queuing capabilities of switches in order to maintain queue occupancies near the ECN marking threshold. Finally, packet pacing is employed for large flows in order to reduce queuing spikes that would widen the variance of queuing delay and would also limit the throughput by increasing the number of packets the PQ marks.

The objective of this project is the implementation of the

HULL architecture on ns-2 (Network Simulator 2) [4] and its end-to-end evaluation with OLDI workloads which the original paper [2] does not cover. The simulation results show that:

- On workloads with SLOs of tens of milliseconds, HULL is a significant improvement over TCP ($\sim 40\%$ lower $99^{th}\%$ percentile latencies) and is equivalent to DCTCP.
- In low latency workloads with SLOs of hundreds of microseconds and fixed processing time at each server, HULL outperforms TCP by a factor of 6 and DCTCP by close to 50% regarding the $99^{th}\%$ percentile latency.
- The $99^{th}\%$ percentile latency improvement is reduced to 11% when variability is added to the per server processing time of the same workload.
- Overall, HULL manages to keep switch queuing latencies in the order of 10 microseconds while DCTCP achieves an order of magnitude higher. Therefore, only very low latency applications can benefit from the few hundreds of microseconds that HULL saves. Finally, the more the variance of the servers' processing time increases, the less the application benefits from HULL.

II. System Design and Implementation

This section describes the three components of the HULL architecture and their implementation in ns-2.

A. Phantom Queues

Phantom queues create bandwidth headroom in order to avoid the need for buffering headroom. The goal is to signal congestion before any queuing takes place. This is in contrast to TCP that relies on packets being dropped by a full queue and to DCTCP that uses a queue occupancy threshold for ECN marking. In the HULL architecture, PQs behave as normal queues that drain at a lower rate than the link's capacity and are placed in series with the egress ports of switches. They do not actually store packets; instead they maintain a counter that is increased upon a new packet arrival and decreased according to the drain rate. If the counter exceeds a threshold, the CE (congestion experienced) IP header field of the packet responsible is set. In this way, the PQ controls the the aggregate rate of the senders so that it is bellow that of the in-series switch port.

The two parameters (drain rate and marking threshold) are tuned in the original paper [2] such that the PQ reacts to congestion quickly and avoids build-up at the switch buffer. The drain rate does not need to be set low to achieve practically zero queuing delay (a baseline of 95% of the link rate is used). In order to facilitate a quick response to congestion, the baseline setting for the marking threshold is $1KB$. The two parameters are inversely related. As the drain rate increases, the virtual queue counter takes longer to reach the threshold and therefore a lower threshold value is needed in order to react to congestion quickly.

For ns-2, the phantom queue was implemented as a component of the SimpleLink class that unidirectionally connects two nodes and simulates packet queuing, transmission and propagation (a switch node does not simulate queuing itself - instead its outgoing links have queues that represent the port buffer). The PQ was placed after the queue object and all dequeued packets go through it. Making sure that only egress switch links have PQs is done at ns-2's OTcl (Object oriented Tcl) [5] front-end. This asymmetry means that two different opposing simplex links (i.e. OTcl SimpleLink objects) are used to connect a switch and a host instead of a duplex link (a duplex link is just a wrapper that creates two identical simplex ones).

B. DCTCP

Using ECN-capable TCP in conjunction with phantom queues would lead to significant loss of throughput because TCP halves its congestion window when congestion is detected - which happens very often when buffering is meant to be minimal - and thus there is no elasticity to absorb bursts of traffic. For this reason, DCTCP is used instead and it operates as follows: The DCTCP sender estimates the amount of congestion on the flow path based on the ratio of ACKs (acknowledgments) that have the ECN-Echo (ECE) field set. Normally, it is an ECN-capable switch that sets the congestion experienced (CE) IP header field of packets that cause the queue length to exceed a threshold. With HULL, the task of setting CE is removed from the switch and assigned to the PQ. In both cases, the DCTCP receiver appropriately sets the ECE TCP header field of the ACKs it sends so that the sender can know how many of the packets it sent were marked. Finally, the sender adjusts its window in proportion to the estimate of congestion it maintains thus avoiding the sawtooth behavior of TCP. The DCTCP algorithm maintains the queue length close to the marking threshold because the correction it does on the window of the sender depends on how much this threshold is violated.

A DCTCP patch for the FullTcp agent of ns-2 is available (see appendix). FullTcp needs to be configured in the OTcl front-end so that it implements the DCTCP algorithm (examples of this can be found in this project's repository - see appendix).

C. Pacing

The use of phantom queues with such small marking thresholds suggests that any flow that sends more than a few packets back-to-back in a burst will receive multiple congestion signals. Additionally, the burst of traffic will temporarily increase the queuing latency at the switch before the sender gets congestion feedback. This will increase the tail latencies of concurrent small workloads. Therefore, in order to reduce queuing spikes and to maintain acceptable throughput for large flows, HULL uses pacing at the hosts to add temporal spacing between the packets of a burst.

Since pacing introduces latency, only non latency-sensitive, throughput-heavy flows are paced.

The HULL pacer is placed after a host's NIC and has two main tasks. The first is to measure the momentary sending rate of the host and slowly adapt its own rate according to the following update rule:

$$R_{tb} \leftarrow (1-\eta) * R_{tb} + \eta * M_r/T_r + \beta * Q_{tb}$$

where R_{tb} is the sending rate, M_r is the number of bytes received in the last T_r seconds, Q_{tb} is the number of bytes that the pacer has in its queue and η, β are positive constants. The pacer updates the rate every T_r seconds. The term $\beta * Q_{tb}$ is needed to increase the rate as the queue length increases.

The second task is to make sure that congestion-causing large flows are paced and small flows are not. To this end, the pacer maintains a flow association table. Initially all flows are not associated. When the the pacer observes an ACK with the ECE field set, it associates its flow with probability p_a (to avoid pacing small flows). The association times out after T_i seconds.

For ns-2, the HULL pacer was again implemented as a component of the SimpleLink class and placed after the queue (represents the NIC of the host in this case) and before the transmission/propagation component. Adding a pacer to a link is again performed in OTcl.

The implementation is as close as possible to the original hardware implementation of a token bucket rate limiter. Whenever a packet arrives from the host, the pacer checks if it belongs to an associated flow. If not, the packet is forwarded with no delay. Otherwise, the pacer checks if there is another packet already queued up, in which case it enqueues the received packet as well (FIFO). If there are no packets in the queue, the pacer calculates whether there are enough tokens available. If there are, the packet is forwarded and its size is subtracted from the number of available tokens. If not, the packet is enqueued and its departure is scheduled based on the current token acquisition rate (R_{tb}). When this scheduled event happens, the next packet in the queue, if it exists, is scheduled similarly.

There are also two asynchronous loops: one that updates the token level (up to a configurable maximum) based on the current rate and one which updates the current rate (R_{tb}) every T_r seconds. When the rate update occurs, the departure time of the packet at the head of the queue is recalculated in order to avoid scenarios in which a packet was scheduled long into the future based on a near zero rate. These asynchronous procedures only add about half a second of runtime for every second of simulation.

Finally, because the pacer sits on the unidirectional link that connects the host to a switch, it does not have access to the ACKs which flow in the opposite direction. This problem is circumvented by the addition of a "gotecnecho" field in the TCP header (existing in FullTcp in ns-2 version 2.34 [6] for the same reason). Outgoing packets use this field to inform the pacer that ECE set ACKs were received. When such a packet is received by the pacer, it associates its flow with probability p_a. If the flow is associated, a disassociation event is scheduled after T_i seconds. This event is rescheduled if the flow gets associated again within T_i seconds.

Network	Link Speed = $1Gbps$, Link Latency = $5\mu s$, Port Queue = $500pkts$
TCP	Max. Window = 1256, Max. ssthresh. = inf, MTU = $950bytes$, Min. RTO = $10ms$, Nagle: Disabled
DCTCP	g = 1/16, K = $30KB$
Pacer	Bucket Depth = $3KB$, η = 0.125, β = 16, p_a = 0.125, T_i = $10ms$
Phantom Queue	Drain Rate = 95%, Threshold = $1KB$

Table I: General System Parameters

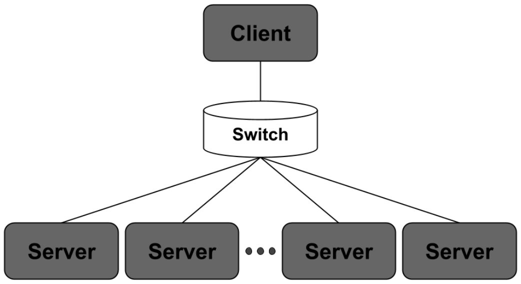
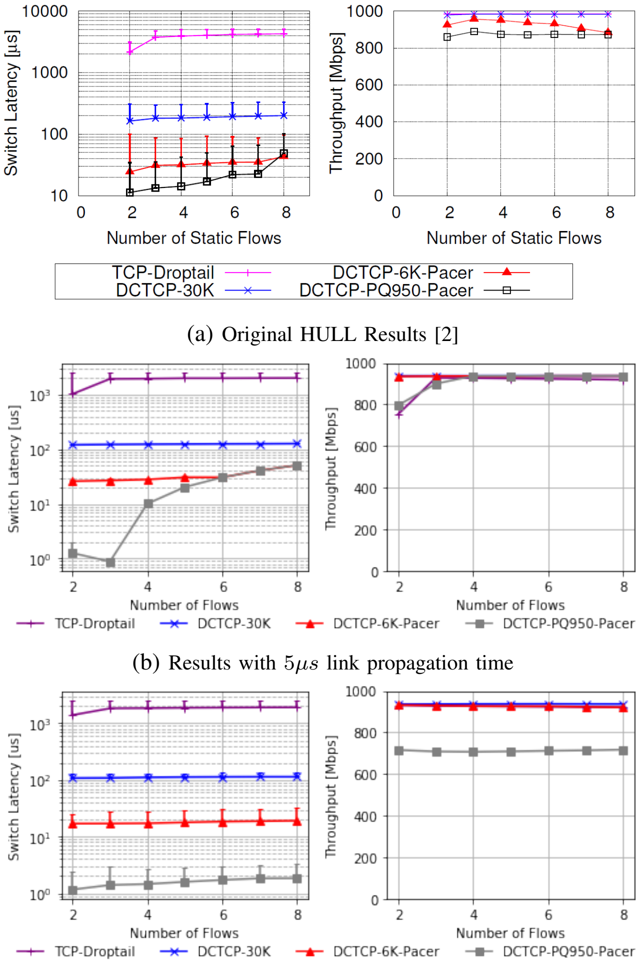


Figure 1: Network topology for all experiments.

III. Results

This section presents the results obtained by simulating the HULL architecture under a variety of workloads using ns-2. First, this implementation is compared to the physical findings of the original paper. Second, the performance of HULL is evaluated under three types of fan-out/fan-in workloads. The architecture is compared to TCP Reno and DCTCP (with the recommended parameters and in a more aggressive configuration). The network topology simulates a single rack with a configurable number of servers connected to the TOR (Top of Rack) switch. In all simulations, one of the servers behaves as a client and issues requests to the other servers (Figure 1). The responses cause congestion at the client's switch port . Table I summarizes the system parameters which hold unless specified otherwise. The MTU is set to $950B$ so that the phantom queues configured with a $1KB$ threshold (as in the original paper) do not mark all maximum sized packets even when there is no other traffic (the PQ increments the virtual queue length upon receiving a packet as described in 5.1 of [2]). The retransmission timeout was set to $10ms$ according to the DCTCP paper [3]. The



(a) Original HULL Results [2]

(b) Results with $5\mu s$ link propagation time

(c) Results with $100\mu s$ link propagation time

Figure 2: Static Flow Experiments (III-A) results in which a fixed number of servers (x-axis) sends data to the client as fast as possible. Left: Queuing latency at the congested client-facing switch port (log scale). Right: Aggregate throughput of all flows. The error bars represent the 99^{th} percentile.

port buffer's fixed size was selected to be similar to the dynamically allocated sizes of the buffers in the original HULL paper.

A. Implementation Evaluation

In order to make sure that this implementation of HULL behaves as intended, two of the experiments of the original paper were reproduced. The results were not expected to be identical as there is a wide range of differences between the original experimental setup and this simulation. Some of those differences are: 1) ns-2 uses fixed size switch buffers compared to the shared, dynamically allocated memory of the Broadcom Triumph2 switch used in the HULL paper, 2) software overhead or features like Interrupt Coalescing are not simulated at the servers, 3) the exact details of the original physical system are not known (e.g. propagation delay, TCP configuration etc.). Nevertheless, the fundamental properties are the same and HULL generally manages to keep the congested port's buffer mostly empty by sacrificing some of the throughput of the large flows.

***Static Flow Experiments**:* This experiment aims to measure throughput and the congested port's queue occupancy under heavy and continuous load. To this end, a client sends requests for the same file to a static number of servers which in turn respond by sending data as fast as possible. The congested port's queue length is sampled every 1ms and is used to calculate the resulting queuing latency. As in the original paper, four configurations are compared: (1) TCP with drop-tail buffers, (2) DCTCP with a $30KB$ marking threshold, (3) DCTCP with a $6KB$ marking threshold along with pacing and (4) DCTCP with both pacers and phantom queues with the marking threshold at $1KB$ and the drain rate at $900Mbps$ (the original paper uses a $950Mbps$ drain rate but it was quite ineffective in this experiment). In configuration 4, it is the PQs and not the switch that are responsible for setting the packets' CE field.

The simulation results can be seen in Figure 2b. In general, HULL reduces the switch-induced latency (i.e. the queue occupancy) by two orders of magnitude compared to TCP and by one order of magnitude compared to DCTCP like in the original results (Figure 2a). The present results differ from the original in the following ways: (1) The complete configuration (DCTCP-PQ950-Pacer) converges to the DCTCP-6K-Pacer at lower flow counts, (2) the latency of the complete configuration is higher with 2 flows than with 3 and (3) the complete configuration has the same throughput as the rest for flows 4 to 8.

The three differences are all caused by the way the $5\mu s$ per-link propagation delay, used in the simulation, affects the round trip time (RTT). Given that the sending rate of TCP-like protocols is approximately $congestion\ window/RTT$ and the RTT is about $40\mu s$ ($20\mu s$ to transmit a 1KB packet twice and $20\mu s$ for the ACK to return due to propagation delays - not accounting for queuing delay), each additional window length unit adds $MTU/RTT \approx 200Mbps$ to the throughput of the flow. The minimum window size used during the simulation was 2 and at this size each flow contributes a minimum of $400Mbps$. With 3 flows, the total of $1200Mbps$ consistently exceeds the drain rate of the phantom queue between the switch and the client and the sending window is fixed at 2 packets. However, the sender practically ignores the marked ACKs as the sending window is already at its minimum size. With 2 flows present, the average window length is 3.4 packets and the senders are probing for additional bandwidth and reacting to congestion normally (hence the higher latency variation).

Figure 2c further illustrates this point. The experiment is repeated with $100\mu s$ link propagation delay instead of $5\mu s$. It is important to note that this is a far-fetched scenario in

		Switch Latency (μs)		900B FCT (μs)		10MB FCT (ms)	
		Avg	**99th**	**Avg**	**99th**	**Avg**	**99th**
20% Load	TCP-Drop-tail	86.2	2284.4	141.7	2387.5	**101.8**	**258.9**
	DCTCP-30K	12.7	241.9	65.2	828.7	102.9	267.2
	DCTCP-6K-Pacer	5.6	170.2	60.8	804.2	104.5	276.9
	DCTCP-PQ950-Pacer	**0.3**	**9.7**	**40.3**	**74.7**	200.5	304.1
40% Load	TCP-Drop-tail	352.3	3412.3	430.3	3560.1	**134.2**	447.8
	DCTCP-30K	48.0	344.9	109.7	965.4	138.9	**423.39**
	DCTCP-6K-Pacer	20.3	225.3	90.9	909.8	143.0	430.8
	DCTCP-PQ950-Pacer	**2.8**	**57.8**	**45.1**	**128.6**	234.9	481.4
60% Load	TCP-Drop-tail	811.7	3718.0	869.3	3791.5	**206.2**	824.2
	DCTCP-30K	102.4	404.8	169.7	897.9	218.3	**724.0**
	DCTCP-6K-Pacer	39.1	255.7	109.7	836.8	226.0	735.9
	DCTCP-PQ950-Pacer	**12.9**	**106.1**	**55.4**	**150.5**	313.3	749.2

Table II: Dynamic flow simulation results. The displayed load level refers to the traffic level on the switch-client link.

which very large flows are synchronized and continuously send data for several seconds. This issue does not affect the rest of the results (at $5\mu s$).

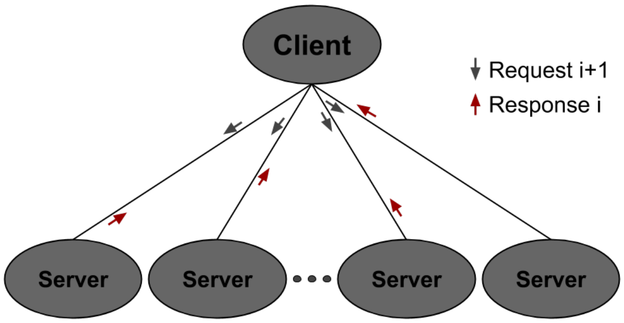


Figure 3: Fan-out/fan-in communication pattern. One of the servers acts as a client/aggregator and sends a request to multiple servers, each containing part of the needed data. Each server responds after some processing time and the client aggregates the responses. The displayed request/response timings are qualitative.

Dynamic Flow Experiments: In this experiment, small $900B$ requests are combined with large $10MB$ ones in order to compare how each configuration performs when it comes to small flow latency and large flow throughput. In this workload, the client sends requests for small and large files at intervals drawn from two different Poisson distributions configured based on the average target load on the congested switch-client link. For each request, the client asks the servers in a round-robin fashion. The ratio between the number of small and large requests is 4 to 1 while the throughput ratio is 1 to $2,222$. The queuing latency at the switch's port facing the client is calculated based on the momentary queue occupancy. Finally, the flow completion times (FCT) are measured for both the small and large flows.

The simulation results for three levels of link utilization are shown in Table II. The results follow a trend similar to the original paper. Considering the 99^{th} percentile of measurements, the simulation of HULL shows an average switch latency reduction of 98.2% compared to TCP and of 82.5% compared to DCTCP (98.3% and 88.6% in the original paper respectively). Concerning the flow completion times of the small flows, the simulation results indicate an average FCT reduction of 96.4% compared to TCP and of 86.8% compared to DCTCP (89.1% and 60.2% in the original paper). Finally, the FCT of the large flows increased by 0.2% compared to TCP and by 8.5% compared to DCTCP (20.3% and 35.8% in the original paper). However, the mean FCT for the large flows increases by 62.2% based on the simulation but only by 34.8% based on the original paper's results.

Given that the simulation results appear to trade-off throughput for latency compared to the original paper, it is reasonable to assume that HULL could be configured less aggressively to account for the differences between the simulation and the physical environment. For example, by setting the PQ marking threshold to $1500B$ for the 20% workload, the switch latency increases from 0.3 to $1.4\mu s$ and from 9.7 to $60.1\mu s$ for the mean and the 99^{th} percentile respectively. At the same time, the equivalent changes for the small flows are from $40.3\mu s$ to $46.5\mu s$ and from $74.7\mu s$ to $362.2\mu s$ and for the large flows the FCTs changed from $200.5\mu s$ to $132.7\mu s$ and from $304.1\mu s$ to $289.8\mu s$. These results are much closer to the ratios observed in the original paper and also illustrate the sensitivity of HULL to the marking threshold.

B. OLDI Workloads

In this section, HULL is evaluated against OLDI workloads. These are fan-out/fan-in workloads in which a client/aggregator assigns parts of a larger request to multiple worker-servers. The client aggregates the server responses and returns the complete result or a partial one if the SLO is violated by some server. An illustration of the fan-out/fan-in communication pattern can be seen in Figure 3. The results that follow correspond to long lived TCP connections and transient phases (e.g. due to the very high initial Slow Start Threshold) are ignored.

Search: First, HULL is evaluated against a web search-like, fan-out/fan-in workload. The client sends $3500B$

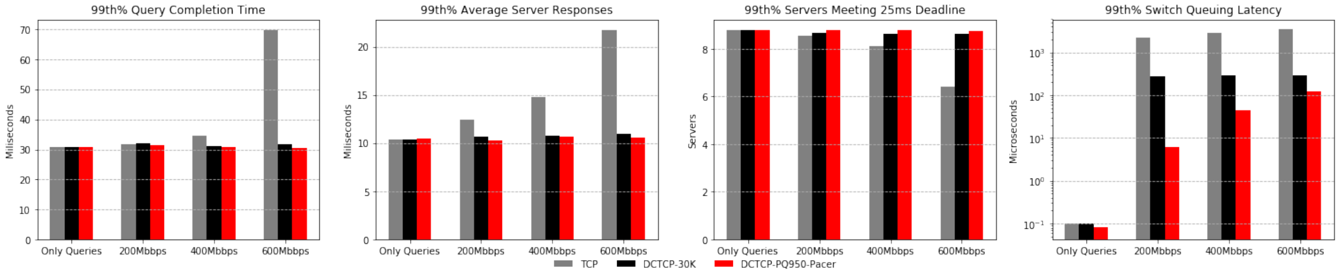


Figure 4: Search workload (III-B) results. An OLDI fan-out/fan-in workload is combined with throughput-heavy background traffic. The x-axis of all plots varies based on the amount of background traffic (from none to 60% of the link's capacity). (a): 99^{th} percentile query completion times (defined by the slowest server response), (b): 99^{th} percentile average server response time (ignores application-level variability), (c): the average of the 99^{th} percentile number of servers that respond within a $25ms$ deadline (worst cases), (d): 99^{th} percentile queuing induced latency at the congested switch-client port buffer.

queries at Poisson intervals to 10 servers. Each server processes the query for a duration drawn from:

$$Service\ time = 180 * gamma(0.7, 20000) + 10000\ [ns]$$

where $gamma(0.7, 20000)$ is a right-skewed $gamma$ distribution with parameters $\kappa = 0.7$ and $\theta = 20000$. The overall mean service time is $2.53ms$ and the standard deviation is $3ms$. Each server's response is $2800B$. A query is considered complete when the slowest server responds. Each server processes the queries sequentially (not a significant factor at this query rate - 30% server occupancy). The workload parameters are based on OLDISIM [7].

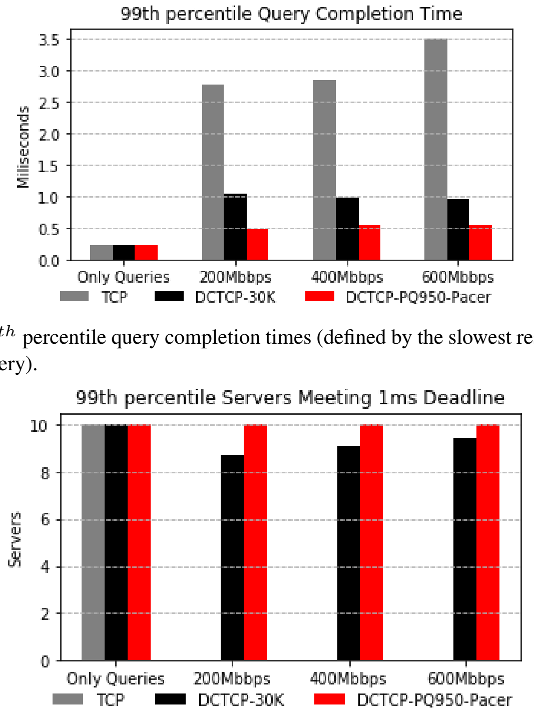
The search workload is also combined with the background dynamic workload described in section III-A in order to simulate a more complete data center environment as described in [3]. The simulation was run at four different levels of average background traffic on the congested $1Gbps$ switch-client link: (1) no traffic, (2) $200Mbps$, (3) $400Mbps$, (4) $600Mbps$. The search workload generated approximately $18,000$ queries and an average of $26.5Mbps$ on the same link.

The results are displayed in Figure 4. Starting from the rightmost chart, when the background traffic is a reasonable $200Mbps$, HULL manages to reduce the 99^{th} percentile queuing latency to $6.1\mu s$ - 46 times lower than DCTCP ($279.1\mu s$) and 355 times lower than TCP ($2163.6\mu s$). However, this sub-$1ms$ reduction compared to DCTCP does not translate to a measurable improvement to the 99^{th} percentile query completion times (leftmost chart) which hover around $30ms$ and are dominated by the tail of the servers' processing time. The same applies to the 99^{th} percentile average server response time which can be seen in the second chart. When it comes to higher levels of background traffic, both DCTCP and HULL outperform TCP on both the 99^{th} percentiles and the means (not shown) by an appreciable margin. The third chart shows the average number of servers which respond within a $25ms$ (in the worst 1% of queries - worst defined by the number of servers that respond) and thus reflects the quality of the result if such a deadline was used. Both HULL and DCTCP have tight tail latencies and the quality is minimally affected by the background traffic.

It is worth mentioning that the pacer successfully selects which flows need pacing. For example, with $200Mbps$ of background traffic, the pacer attached to one of the servers decided to associate the search flow 2542 times out of the 72000 search packets (3.5%) and the background flow $18,106$ times out of the $390,000$ packets (4.6%). This count includes the times a flow was already being paced - in that case the disassociation timer was reset. The difference between the two flows is that the search flow was disassociated 2091 times (82%) while the background flow was only disassociated 139 times (0.8%). Essentially, even if a search query was paced, the next one was unlikely to be so while every background $10MB$ transfer was paced.

Low Latency Workload with Constant Service Time: The conclusion that can be drawn from the previous workload is that, while HULL outperforms TCP, the additional few hundreds of microseconds of queuing time saved compared to DCTCP need an appropriate, very low latency application, in order to have an impact. In this section, the performance of HULL is evaluated against a low latency fan-out/fan-in workload without service time variability. It is similar to the previous workload but the size of the queries and the responses is now $100B$ and critically, the per-server service time is fixed at $100\mu s$. The workload coexists with background traffic as before. Approximately 50000 queries are generated along with an average of $4Mbps$ on the congested switch-client link.

The results are presented in Figure 5. Across the scenarios that involve background traffic, HULL reduces the 99^{th} percentile query completion time by 83% compared to TCP and by 47.3% compared to DCTCP. Additionally, HULL offers perfect response quality with a strict $1ms$ response deadline.



(a) 99^{th} percentile query completion times (defined by the slowest response per query).

(b) 99^{th} percentile number of servers that respond within a $1ms$ deadline (worst cases).

Figure 5: Very low latency workload (III-B) results. A fan-out/fan-in workload with deterministic processing time per server of $100\mu s$ is combined with different levels of background traffic. The x-axis varies based on the volume of background traffic (from none to 60% of the link's capacity).

The Effect of Application-Level Variance: In this section, the previous experiment is repeated but now: 1) it is scaled to a full rack (40 servers - 1 fan-out/fan-in flow per server) and 2) each server takes a random amount of time to respond drawn from:

$$Service\ time = 180 * gamma(0.7, 790) + 400\ [ns]$$

where $gamma(0.7, 790)$ is a right-skewed $gamma$ distribution with parameters $\kappa = 0.7$ and $\theta = 790$. The overall mean service time is the same as in the previous workload at $100\mu s$ and the standard deviation is $119\mu s$. Applications such as in-memory databases often display variability in their response time [8]. Therefore, this is a more realistic scenario in which both network and application-level latency variability are considered. The results are shown in Figure 6. HULL outperforms DCTCP in the 99^{th} percentile query completion times (leftmost plot) by only 11%. This is due to the randomness in the time a server takes to process the query. Because a query is considered complete when all the worker servers have responded, the improvement HULL brings is reduced. This is illustrated in the middle plot where the 99^{th} percentile average server response time is depicted. This metric ignores the application-level randomness and as a result, HULL maintains its $\sim 50\%$ advantage over DCTCP.

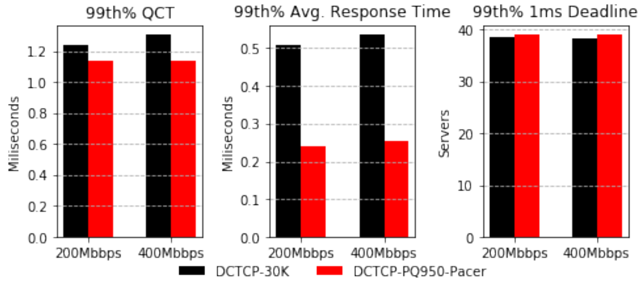


Figure 6: Results for the very low latency, variable server processing time workload (III-B). The same workload as in Figure 5 but using random server processing times. The x-axis varies based on the amount of background traffic. Left: 99^{th} percentile query completion time, middle: 99^{th} average server response time (ignores application-level variance), right: the average number of the 99^{th} percentile of servers that meet a $1ms$ SLO (average of worst cases).

IV. Conclusion

The goal of this project was the implementation and end-to-end evaluation of the HULL architecture. HULL was implemented on ns-2 and the simulation results from experiments similar to those carried out on a real topology by the original paper authors are qualitatively similar. HULL was evaluated against user-facing, fan-out/fan-in workloads in order to measure the effect it can have on the user experience. HULL offers great tail latency improvements compared to TCP in all tested scenarios in which background traffic is present. It only outperforms DCTCP in very low latency applications (SLOs of hundreds of microseconds) but the reduction in the 99^{th} percentile query completion time gets less significant as the application-level response latency variance increases.

References

[1] J. Dean and L. A. Barroso, "The tail at scale," *Commun. ACM*, vol. 56, no. 2, pp. 74–80, Feb. 2013. [Online]. Available: http://doi.acm.org/10.1145/2408776.2408794

[2] M. Alizadeh, A. Kabbani, T. Edsall, B. Prabhakar, A. Vahdat, and M. Yasuda, "Less is more: Trading a little bandwidth for ultra-low latency in the data center," in *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 19–19. [Online]. Available: http://dl.acm.org/citation.cfm?id=2228298.2228324

[3] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan, "Data center tcp (dctcp)," in *Proceedings of the ACM SIGCOMM 2010 Conference*, ser. SIGCOMM '10. New York, NY, USA: ACM, 2010, pp. 63–74. [Online]. Available: http://doi.acm.org/10.1145/1851182.1851192

[4] “ns-2,” https://www.isi.edu/nsnam/ns/.

[5] “Otcl,” https://en.wikipedia.org/wiki/OTcl.

[6] “ns-2 version 2.34,” https://sourceforge.net/projects/nsnam/.

[7] “Oldisim,” https://cloudplatform.googleblog.com/2015/03/benchmarking-web-search-latencies.html.

[8] G. Prekas, M. Kogias, and E. Bugnion, “Zygos: Achieving low tail latency for microsecond-scale networked tasks,” in *Proceedings of the 26th Symposium on Operating Systems Principles*, ser. SOSP ’17. New York, NY, USA: ACM, 2017, pp. 325–341. [Online]. Available: http://doi.acm.org/10.1145/3132747.3132780

Appendix

Simulating HULL on ns-2

A. *ns-2 and DCTCP*

HULL was implemented for ns-2 2.34. Straightforward instructions on installing ns-2 on Ubuntu and adding DCTCP can be found `here`.

B. *Adding HULL to ns-2*

The modified ns-2 files can be found in the project’s `repository`. After installing ns-2.34, updating the environment variables and patching-in DCTCP, do:

- Replace ns-2.34/tcl/lib/ns-default.tcl (or add lines 413-414 and 692-702)
- Replace ns-2.34/tcl/lib/ns-lib.tcl (or add lines 1569-1572)
- Replace ns-2.34/tcl/lib/ns-link.tcl (or add lines 540-559)
- Replace ns-2.34/link/delay.h and delay.cc
- Add hull-pacer.h and hull-pacer.cc to ns-2.34/adc/
- Add ”adc/hull-pacer.o” to the OBJ_CC variable of ns-2.34/Makefile.in
- In ns-2.34/ do: `./configure`, then `make clean` and then `make`.

C. *Reproducing the results*

Note: the simulation result files are available in the repository and can be used to avoid running the simulation scripts which in some cases take a day or two to complete.

Reproducing Figure 2b (static flow):

1) Run ”static_flow_main_coord.sh” to run all the relevant experiments (takes 10s of minutes).
2) In results/static_flow_main (result folder for this script) run the ”postprocess.ipynb” notebook.

Do similarly for Figure 2c by running ”static_flow_100_coord.sh” and then ”results/static_flow_100/postprocess.ipynb”

Reproducing the data from Table II (dynamic flow):

1) Run ”coordinator_dynamic.sh”. Folders named ”results/nofanout_(load_level)_9flows” will be created and populated (takes minutes).
2) Run ”results/parse_final_coord_dynamic_wrkld.ipynb”.

Reproducing Figure 4 (search workload):

1) Run ”coordinator_search.sh”. Folders named ”results/bkg_(load_level)_fanout_wkld1_30_10flows” and ”results/onlyfanout_wkld1_30_10flows” will be created and populated (takes hours).
2) Run ”results/parse_final_coord_fanout.ipynb”.

Reproducing Figure 5 (deterministic low latency workload):

1) Run ”coordinator_100B.sh”. Folders named ”results/bkg_(load_level)_fanout_wkld0_5_10flows” and ”results/onlyfanout_wkld0_5_10flows” will be created and populated (takes days).
2) Run ”results/parse_final_coord_fanout_100B.ipynb”.

Reproducing Figure 6 (random low latency workload):

1) Run ”coordinator_100B_gamma.sh”. Folders named ”results/gamma_bkg_(load_level)_fanout_wkld6_3_40flows” will be created and populated (takes days).
2) Run ”results/gamma_100B_low-latency.ipynb”.