A Novel Transport Protocol for Network Telemetry Collection

Authors

1. ABSTRACT

The emergence of programmable switches makes it possible to collect a large amount of fine-grained telemetry data in real time in large networks. However, transferring the data to a telemetry collector and relying on the collector CPUs to process telemetry data is highly inefficient. RDMA is commonly used as an efficient data transfer protocol but it does not work for telemetry data. This is because it is challenging to create RDMA format packets at switches, support lossy networks, and support various memory access patterns such as multiple concurrent writes. In this paper, we introduce a new direct telemetry data access system, which allows fast and efficient telemetry data transfers between switches and the remote collector. Our system introduce telemetry report primitives such as Key-Write, Append, Aggregation, and Key-Increment. To support these primitives and address the limitations of RDMA, we use some programmable switches as translators that generate RDMA packets, handle packet losses, and aggregate diverse memory accesses. Our evaluation demonstrates that we can enhance existing telemetry framework by XX%.

2. INTRODUCTION

ToDo: ***Write an introduction***

3. BACKGROUND?

——— SR: Content below not ready yet

Networks today are growing in size to support large amounts of traffic and network operators are keen on gaining more insights into the network via monitoring to improve performance for their network. However, monitoring all events at all times can be challenging for two main reasons. First, collecting all events generated from the network can be slow as processing reports at the collector can be time-consuming. We call this as the *collection bottleneck*. Second, after collecting the reports, querying the reports to gain insights on the network can be slower as the reports may not be stored in an optimal manner for queries. We call this as the *querying bottleneck*.

The collection bottleneck arises on how the collector receives and processes the reports. At the collector, CPU cycles are allocated to receive the report (referred as I/O) and operations are done on the report for storage (referred as storage). Typically, the storage operations involve parsing the reports to extract monitoring information, filtering the reports to remove redundant reports and determining the memory location for storing the report.

Existing works may use a combination of strategies to tackle the collection bottleneck as shown in Figure 2. For this setup, we generated SR: x reports and used the traditional INTCollector, Confluo and DART to see the amount of time it takes for I/O and storage operations. For instance, the traditional INT collector uses raw sockets for collection and stores reports in memory.SR: How does the INT collector perform filtering/indexing? Trying to get the bottleneck here. However raw sockets with regular storage of reports SR: Looking for a term here – basically something that happens on usual cases where you have to store something in memory. can take significant CPU cycles for I/O and storage. Confluo, improves on the INTCollector by using DPKD instead of raw sockets and uses an atomic multi log to optimize storage. Use of DPDK reduces the I/O cycles and storage by a factor of SR: x times– y times. However, it still depends on CPU at the collector to determine memory locations to store consumes many CPU cycles. DART uses RDMA to write reports to the collectors memory by using programmable switches that determine the memory location to bypass the CPU of the collector. The use of RDMA further reduces the number of cycles for storage by SR: x times. Therefore, RDMA

The querying botteleneck arises due to the lack of organization of reports in memory. A typical query would involve the collector to determine the indexes where the reports are stored, retrieve the stored data, perform operations on the retrieved data and return the results in the requested format. Figure, shows the CPU cycles needed for performing SR: a,b,c queries.SR: Come back after this is complete.SR: ask gabrielle

SR: I think we need a graph to show the CPU cycles for a query to set us apart from Confluo and DART. We should also have a bar for "optimised indexing", where we show how much CPU cycles it takes if we use key-write/append etc

Ideally, a collector can leverage the benefits of RDMA along with better storage primitives to reduce the collection and storage bottleneck. In this paper, we propose a new transport protocol, that leverages programmable switches to generate custom RDMA packets that allow the collectors to store reports that could be easily queried.

4. NEED FOR BETTER STORAGE PRIMITIVES

SR: - Is it 50ms over 500ms? What is the big difference? - Play with the data injection –> That is the location memory. Understanding where to find it. - Focus on the scalable aspect of the collector – CPU – Lets remove the CPU – how to remove RDMA. - How can we leverage we use x,y primitives? point it to the section of the challenges – SR: I think we should shift the current table 2(storage primitive) here, and also the content for the primitives (compressed) here. The idea is after we show that this primitives are useful to implement, we can highlight the challenges of using RDMA to implement them.

5. CHALLENGES

Designing telemetry collection around RDMA has the potential for significant performance improvements. However, RDMA and related protocols impose several restrictions on such a design:

RDMA is limited to basic memory operations Achieving the full performance potential of RDMA requires restricting ourselves to one-way verbs, i.e., pre-defined primitives that execute entirely on the network card without requiring CPU intervention. This limits us to basic memory operations: *Read*, *Write*, *Fetch-and-Add*, and *Compare-and-Swap*. Restricting the design to these fundamental primitives greatly reduces the viability of dynamic data structures such as linked lists and cuckoo hashing. SR: Can we motivate why we need complex data structures and why not a simple store not enought?Such data structures require us to first read memory at the collector before knowing exactly what to write where, leading to significantly more complex on-switch logic. Further, one must ensure that there are no conflicts or race-conditions occurring between different switches that work in parallel to report telemetry information.JL: We need to make it super clear that we can deliver new primitives that standard RDMA couldn't feasibly do. E.g., in the case of Append, how would they synchronize the head pointer?

A limited number of RDMA connections RDMA network cards have limited memory, which constrains the number of active connections that fit in their local cache. Having more simultaneously active RDMA connections to a collector than can fit in the cache forces the network card to start swapping connection statefulness to server memory, which *significantly* reduces RDMA performance [6]. It is therefore recommended to keep the number of queue pairs low to ensure maximum performance [14]. Potential workarounds such as having several switches share the same connection become very complicated, due to the RDMA requirement of sequential packet identifiers within each connection, which would require sharing switches to synchronize these values on each new telemetry report.

High-speed RDMA assumes a lossless network Another challenge related to the packet identifiers is handling network losses. Because RDMA assumes that incoming packets in a connection have sequentially increasing packet identifiers, even just a single lost RDMA packet would result in the RDMA connection state invalidating due to desynchronized packet identifiers between the switch and collector, which leads to the network card discarding every subsequent RDMA operation until the switch actively manages to resynchronize the connection.

Managing RDMA connections is costly for switches RDMA traffic is not designed for ease-of-generation in network switches. For example, the RoCEv2 protocol, which is the standard for RDMA communication for Ethernet, is especially costly. This protocol requires a relatively high amount of statefulness on a per-connection basic for tracking essential metadata, significant header overhead to calculate and generate, and a large footprint on internal busses handling RoCEv2-imposed checksumming. SR: Can you give an example? Something like, a single connection costs X MB, takes x amount of time to process. This is y% of the entire memory available. Requiring such a footprint on every single telemetry-reporting switch in a network is certainly doable, but very inefficient.JL: remove?

The data must be queryable Allowing efficient centralized access to telemetry data is the main reason for centralized collection. How to make it queryable? Cross-switch collaboration etc..JL: Too similar to first?SR: I am not sure if this is a problem with RDMA. Isnt it a general collection problem?JL: remove

SR: Content above not ready yet

5.1 What is RDMA?

JL: Explain RDMA

Queue-pairs == connections PSN One-way vs two-way Available verbs RoCEv2

6. MOTIVATION

JL: This should build on background and dig into why CPU-based collection is inefficient. JL: Waiting for Gabriele results before planning structure

MY: Give background on classes of telemetry systems, and draw a table on their common requirements/bottlenecks on data collection/query.

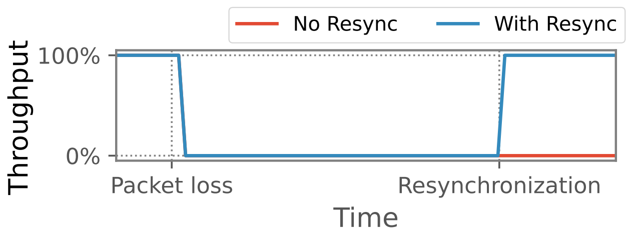


Figure 1: RDMA connections require resynchronization after packet losses occur, or the NIC will discard incoming packets.

System	Per-switch Report Rate
INT (non-sampling)	4.75 Gpps
INT (0.5% sampling)	23.75 Mpps
Marple	950 Kpps
NetSeer	4.29 Mpps

Table 1: Per-reporter data generation rates by various monitoring systems. Assuming 6.4Tbps switches JL: I want something like this in motivation, but maybe not in table format?

Collectors play an important role in network telemetry systems: they receive telemetry reports and store the information in an internal data structure to be used to answer network-wide queries. One key challenge is to ensure that this process is scalable as a datacenter network can comprise hundreds of thousands of switches [11]. For example, a non-sampled INT telemetry system requires the collection of telemetry data from *every single packet*, which would result in an excessive amount of reports. Because of this, event detection is typically implemented at switches in an effort to send reports to a collector only when things change [13]. This helps in reducing the rate of switch-to-collector communication down to a few million telemetry reports per second per switch [24], at the cost of reduced network insight and increased on-switch complexity. Still, telemetry collection costs are high, and the main reason we identified is that the collectors' CPU is the main bottleneck. SR: I think it would be good to break down what steps that goes into collection. For instance, 1) a packet is generated from switch (or multiple switches), 2) packet is then routed to a collector, 3) at collector appropriate fields are read 4) collector then stores 5) some form of bookeeping is done to maintain large amounts of collected data. Then, we should say, we find steps 3,4,5 to be the bottleneck and move onto 3.1, 3.2 and 3.3. Can we also talk about potential race conditions here?

6.1 Monitoring Systems are Intense

JL: Show some example numbers on the amount of telemetry repots generated.

6.2 CPU-based collection is inefficient

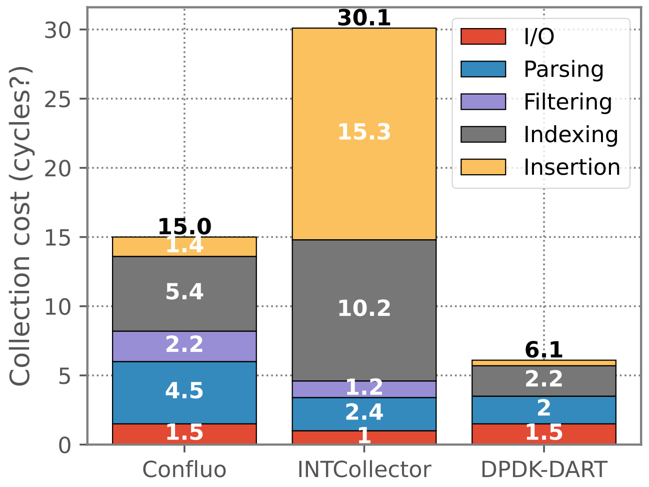


Figure 2: Breakdown of telemetry collection costs in CPU-based solutions. DPDK-DART is functionally equivalent to DTA Multi-Write (This is placeholder data!)JL: this will only be a breakdown of Confluo, and likely more detailed

MY: key bottleneck or the ignored bottleneck in many telemetry systems today

MY: Discuss existing monitoring solutions

JL: Refer to table 1. Put this in comparison to benchmarked performance of Confluo

We show examples of what the CPU spends time on in Figure 2, and discuss how they already attempt to optimize these numbers. DTA would either bypass or offload these steps into hardware.

6.3 RDMA enables high-speed memory writes

What if we design telemetry collection around RDMA?

7. DTA OVERVIEW

DTA achieves scalability, generalizability, and high performance by decoupling telemetry reporting switches from the underlying storage backend. Reporters simply send a DTA packet, containing information about the telemetry data, towards one of the deployed telemetry collectors. These DTA packets are intercepted by the last hop towards the collector, by the top-of-rack switch. This last switch acts as a *DTA translator*, and is responsible for making the telemetry data queryable at the collectors in this rack.

The translator parses incoming DTA packets, and converts these into suitable RDMA operations that are forwarded to the right collector. Generated RDMA traffic is determined by the incoming DTA packet, which specifies the specific DTA primitive that is requested, together with essential DTA metadata according to the needs of the primitive (please see section 8 for a discussion on DTA primitives).

7.1 Translation Benefits

A naive RDMA-based telemetry collection design might generate RDMA traffic immediately at the telemetry gener-

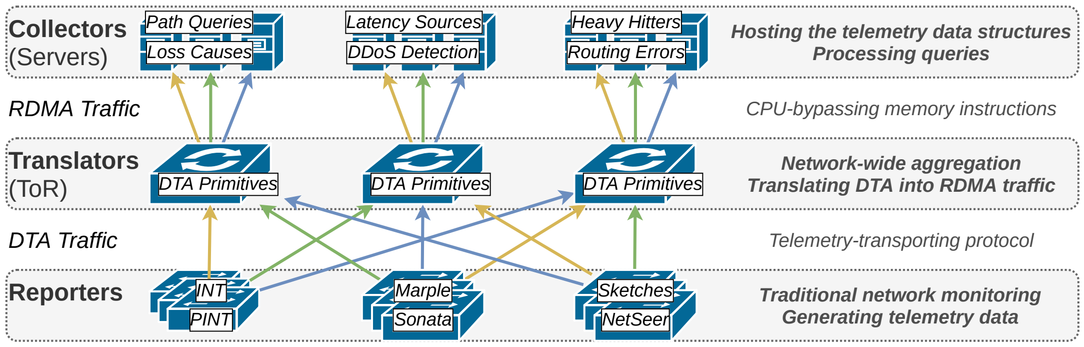


Figure 3: Overview of the DTA telemetry reporting flow JL: Add "traditional" flow alongside? With TCP/UDP to collectors

ating network switches (i.e., at the reporters), which would require determining a storage server and a memory address inside of that server, and then generating an RDMA call to update the telemetry storage.

Introducing a DTA translator to intercept DTA traffic allows for several benefits compared to directly generating RDMA from individual switches:

Expanded RDMA with new functionality A translator is guaranteed to be the last processor for all telemetry packets destined to the collectors that it manages. This allows us to expand RDMA with new primitives and functionality that take advantage of a central viewpoint, including the addition of the powerful *Append* and *Sketch-Merge* primitives, centralized data duplicate detection and filtering, and real-time detection of network-wide events.

Improved collection speeds The translator is in full control of the RDMA state of each collector, which allows performance improvements compared with designs where each reporter would handle their own RDMA communications directly. All telemetry reports towards a collector can now be combined into just a few RDMA connections which are maintained by the translator, which improves system performance by reducing queue-pair swapping in the RDMA NIC[6, 14].

Increased system stability RDMA expects network losslessness to achieve its high-performing potential, and a single RDMA packet lost in transit results in an invalid RDMA queue-pair state and a complete loss of throughput until the two RDMA end-points actively resynchronizes. A single switch can now detect and preempt RDMA congestion during periods of spiking telemetry data, and significantly reduce the overheads and delays of resynchronization if the NIC still fails to process a generated RDMA packet, since only a single and directly attached device has to resynchronize the connection.

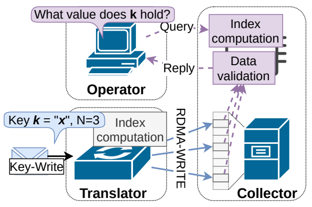
Reduced in-network costs Managing RDMA states and packet generation inside of the switch ASIC is costly. Introducing the translators allows us to strip this footprint from ordinary telemetry-generating switches that exist all across the network, keeping this functionality and footprint inside only a few specialized collector-managing top-of-rack switches. Telemetry reporters would now only have to generate reports through the much more lightweight DTA protocol, designed specifically for ease of in-ASIC generation (see section 10.6 for a cost comparison).

7.2 ToR vs SmartNIC Dilemma

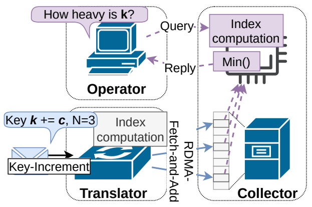
We have concluded that the addition of a DTA translator as the last hop before collection would be beneficial in several ways. The question is then *how* to deploy this translator, where we could take one of two approaches. One could leverage programmable network cards to build in DTA support to the actual network card that is installed at the collector. The NIC could then process incoming DTA operations, and translate these not into RDMA, but directly into *DMA* operations that are immediately sent through the PCI bus to memory. This would allow for faster memory operations compared with a switch approach, and might even make stateful solutions that leverage server DRAM more feasible. One example might be to include collision-mitigating techniques such as Cuckoo Hashing for the Key-Write primitive, or the addition of more powerful aggregation functions alongside Key-Increment.

However, highly programmable network cards often suffer from reduced performance compared with fixed-function network cards [9], and our evaluations show that the ToR-as-translator approach already results in the NIC message rate being the collection bottleneck (§10.1.1 and §10.2), even while using top-of-the-line fixed-function RDMA NICs. Not even upcoming P4-programmable ASIC-based NICs manage to deliver packet processing rates close to what current-generation fixed-function RDMA-capable NICs deliver [20]. The exception is FPGA-based network cards, which deliver near fixed-function ASIC performance, and are already deployed in hyperscale data centers to a limited extent [7]. FPGA-based NICs are however famously power hungry, expensive, and difficult to program [9] which limits their adoption.

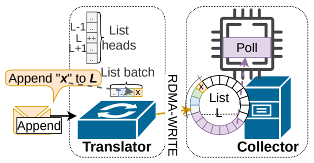
We concede that programmable NICs would result in smaller infrastructure changes than using programmable ToRs, since modifications would only have to take place at the server-level. The DTA design presented in this paper is designed



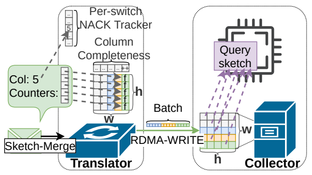
(a) DTA Key-Write



(b) DTA Key-Increment



(c) DTA Append



(d) DTA Sketch-Merge

Figure 4: A high-level overview of the proposed DTA primitives

around highly pipelined architectures, and we therefore expect that the design choices of our system would translate well into most programmable packet processing architectures that might exist in future network cards.

7.3 RDMA Connection Stability

RoCEv2, which is the de-facto standard for RDMA in Ethernet-based networks, includes packet sequence numbers (PSNs). Each end-point in an RDMA connection tracks the PSNs of packets in both directions, first to make sure that it itself generates sequentially incrementing PSNs, and to detect non-sequential incoming RDMA operations for example due to system overload. RDMA is designed under the assumption of lossless networks, and just a single lost packet will lead to complete loss of RDMA throughput due to the PSNs that are generated at one end and expected by the other becoming desynchronized.

A desynchronized queue-pair has to be actively resynchronized so that the two end-points once again agree on the expected PSNs in the traffic flows. Queue-pair resynchronization is achieved in DTA by including in-translator support for processing RoCEv2 acknowledgements that are coming from the collector NIC, which are generated once immediately when an incorrect PSN is detected.

The translator can then use information provided by this acknowledgement to reset the PSNs that the switch includes in RDMA packets. Successful resynchronization also requires a grace period where RDMA traffic is entirely prevented, to allow the NIC to clear internal buffers of invalid traffic. Failure to allow the NIC to clear buffers, either due to a too short or non-existant grace period, will lead to immediate desynchronization once again.

8. DTA PRIMITIVES AND "GENERALIZEABILITY"

Different monitoring systems each put their own requirements on a potential collection system. Telemetry collection designs therefore have to present several different primitives, to fulfil the collection needs of each system in a heterogeneous monitoring environment. DTA presents four different collection primitives that together enable support for a wide range of different monitoring systems: *Key-Write*, *Append*, *Sketch-Merge*, and *Key-Increment*, and shown in Figure 4. Table 2 shows examples of how current state-of-the-art monitoring systems can be integrated with DTA-based collection.

8.1 Key-Write

The Key-Write primitive (see Figure 4a) is designed to allow for key-value storage in DTA, where each piece of telemetry data has a unique pre-known key that can be used for retrieval of the reported information. Key-value storage, and hence the Key-Write primitive, is especially useful in cases where dynamic identifiers are preferred, such as when we want to report information on a per-flow basis. Identifiers for this information would include some sort of flow identification such as the 5-tuple, and would therefore be difficult to efficiently map into fixed, pre-allocated, and unique identifiers. Per-flow data and queries is but one scenario where the Key-Write primitive is useful, and we present several examples in Table 2.

Telemetry reporters would only have to send a single DTA packet towards one of the collection racks, containing the *key*, *telemetry data*, and a *redundancy* integer which will be further explained in section **??**. The DTA translator will parse this Key-Write request, and generate RDMA traffic to insert this data into a global key-value store in an adjacent server. See section 9.2.2 for details of how DTA implements Key-Write in hardware.

Our low-level realization of the Key-Write primitive is fundamentally based around DART [16], which has been extensively evaluated in terms of data robustness, reliability, and memory efficiency. This algorithm is built to deliver probabilistic key-value storage of telemetry data, and is designed for efficient data plane deployments. It achieves this by constructing a central key-value store that acts as a shared hash table for all telemetry-generating network switches. Indexing of per-key data in this hash table is performed statelessly through global hash functions without collaboration, which means that there is nothing preventing different keys from hashing into the same locations. The algorithm therefore inserts telemetry data as N identical entries at N different memory locations to achieve partial collision-tolerance through built-in data redundancy. A checksum of the telemetry key is stored alongside each data entry, which allows queries to be verified by validating the checksum.

DTA allows switches to specify the *importance* of per-key telemetry data by including the level of redundancy as a field in the Key-Write header. This prevents data of low importance from having as high occupancy in the shared data structure, at the cost of reduced data queryability due to a lowered collision tolerance as shown in section 10.1.2. The level of

Primitive	Example monitoring	Description
Key-Write	INT (Path Tracing) [10, 15]	INT sinks reporting 5x32b switch IDs using flow 5-tuple keys
	Marple (Host counters) [18]	Reporting 32-bit counters using source IP keys, through non-merging aggregation
	Sonata (Per-query results) [12]	Reporting network query results using queryID keys
	PINT (Per-flow queries) [2]	PINT can leverage the memory redundancies for improved data compression through $n = f(pktID)$
	PacketScope (Flow troubleshooting) [21]	Report per-flow per-switch pipeline traversal information using <switchID,flow 5-tuple> as key
Append	INT (Congestion events) [10, 15]	INT sinks report a period of network congestion to list of network congestion events
	NetSeer (Event aggregation) [24]	Sorting per-switch events into network-wide lists according to category
	Sonata (Raw data transfer) [12]	Sending selected raw data from switches to streaming processors, to be the base for query responses
	PacketScope (Pipeline-loss insight) [21]	On packet drop: send pipeline-traversal information to central list of pipeline-loss events
Sketch-Merge	C [3] and CM [5] sketches	Counter-wise `sum` of the sketches of all switches
	HyperLogLog [8, 4]	Register-wise `max` of the sketches of all switches
	AROMA [1]	Select network-wide uniform packet- and flow samples from switch-level samples
Key-Increment	TurboFlow (Per-flow counters) [19]	Sending evicted microflow entries for central aggregation using flowKeys as keys
	Marple (Host counters) [18]	Reporting 32-bit counters using source IP keys, through addition-based aggregation

Table 2: Example telemetry monitoring systems and scenarios, as mapped into the primitives presented by DTA

redundancy that was included at report-time is not known while querying, and the collector will therefore default to an assumed maximum redundancy level of $N = 4$ and process all non-used redundancy slots as overwritten entries.

We significantly reduce the network overheads of DART by moving the indexing and redundancy generation into the DTA translator. This design choice effectively reduces the network telemetry traffic by a factor of the level of redundancy, and further reduced the telemetry report costs in the individual switches by replacing costly RDMA generation with the much more lightweight DTA protocol. See section 9.2.2 for an in-depth explanation of the we implement the Key-Write primitive in hardware.

8.2 Key-Increment

The DTA Key-Increment primitive (see Figure 4b) is fundamentally very similar to the Key-Write primitive, with one main exception: it allows for addition-based data aggregation. What this means is that the Key-Increment primitive does not instruct the collector to set a key to a specific value, but it instead *increments* the value of a key. This can be useful for several scenarios, especially when aggregating network counters. The simple Key-Write primitive would not allow for collection of distributed counters, where different switches increment the same key, or when switches lack the amount of memory that is requires to keep all counters in-memory. For example, switches might only store a few counters in a local cache, and evict old counters from the cache periodically when new counters take their place [19, 18]. The Key-Increment primitive can then deliver collection of these evicted counters at RDMA rates.

This leads to two main design changes compared with Key-Write. First, we do not write the incoming data into N locations. Rather, we increment the N pre-existing values through the RDMA Fetch-and-Add primitive. Second, the data can not be queried through the Key-Write algorithm due to collisions likely resulting in all N values containing some value error. We leverage properties of Count-Min Sketches [5], and query Key-Increment data by retrieving all N memory entries and returning back the *lowest* of these as an answer to the query. JL: Ran you might want to rewrite this :)

The introduction of a translator can not only improve on the Key-Increment algorithm through reduced network overheads compared with non-translating designs, but can also allow for central caching of incoming counters. Such caching can trigger RDMA generation only when a cached counter is *evicted* from the cache either by becoming idle (and being pushed out by new counters) or having been cached too long (preventing stale counters).

8.3 Append

Some telemetry scenarios can not easily be collected through a key-value store. One such reason is if there are not obvious keys can can be pre-known and unique for all telemetry data in the system, or if the underlying telemetry data is not fixed-size but can grow over time. An example is if one wants to report high packet loss events in the network. These reports could include an identifier for where a packet of set of packets were lost, to allows for controller troubleshooting and reactivity. Simply using a key-value storage for this would either just overwrite old data if a single key is used, to significantly reduce queryability by forcing collectors to iterate over a large list of possible keys to find the ones which actually has been used to report information.

A more streamlined collection solution is to allow for reporters to append information into global lists containing a pre-defined telemetry category in each list. Network operators could then allocate dedicated lists for the types of telemetry data that they extract from the network. Examples of interesting lists could be *congested links*, *packet loss events*, *latency spikes*, or *suspicious flows*.

Telemetry reporters simply have to craft a single DTA packet to declare what data they want to append to which list, and forward it to a collection rack that hosts this list. The translator would then generate an RDMA packet that inserts this new telemetry data in the correct slot in the pre-allocated list. See section 9.2.3 for details of how DTA implements Append in hardware.

The Append primitive as a concept would not have been feasible with a naive RDMA-based collection system where every telemetry-reporting switch simply generated RDMA traffic directly. Appending data into memory requires knowledge of where to write relative to the start of the memory buffer, to allow sequential data insertions. Such a non-translating design would either require dedicated lists on a per-switch basis, or somehow synchronize the memory pointers that are used to track where to write telemetry information across all switches.

DTA translation instead allows us to keep a single in-network pointer, which is located at the translator. This allows us to insert incoming per-list Append reports sequentially and contiguously into memory, which leads to a very efficient use of memory and impressive query performance as shown in section 10.3.

Further, translation allows us to *significantly* improve on the collection speeds by batching multiple reports together, as explained in section 9.2.3 and demonstrated in section 10.2.

8.4 Sketch-Merge

Sketches are a popular mechanism to summarize the traffic going through a switch using a small amount of memory and with provable guarantees. Common problems that are solved with sketches include flow-size estimation (e.g., C [3] and CM [5] sketches), estimating the number of flows (e.g., HyperLogLog [8]), and finding superspreaders (e.g., [1]).

An essential property of many of these sketches is *mergability*. Broadly speaking, one can take the sketches of individual switches and merge them to obtain a network-wide view. Merging procedures differ between sketches; for example, the C and CM sketches require counter-wise summation while Hyperloglog needs to do register-wise `max`. Existing solutions commonly report the sketches to the controller at the end of each epoch (e.g., [17]) which merges them together in CPU. However, this is a costly operation that we wish to avoid using RDMA. While some sketches are directly mergable using existing RDMA primitives (e.g., Fetch & Add), we argue that it is better to do the aggregation at the translator for several reasons:

- It can support merging procedures that RDMA cannot such as `max`.
- It can RDMA the aggregation result using a small number of writes, thus reducing the NIC and memory overheads at the collector.
- It is the first point to observe all data, thus translator-aggregation would reduce the network overheads.

Accordingly, we design a translator-aggregation solution that is efficient and robust to packet loss. First, we need the switches to send their sketch; this problem is not specific to our solution, and we can use a similar approach to LightGuardian [23] where we send one or more columns in each packet. Unlike LightGuardian, every switch sends the columns *sequentially*, and the translator keeps track of the last column received from each switch. If, for example, column 7 arrived while the last column processed was 5, the translator notifies the switch using a NACK packet, and the switch starts over from column 6. Next, the translator tracks the number of sketches it aggregated *per column*. Once a column has received counters from all switches, it is ready to be written to collector memory. For efficiency, our solution batches several columns together into a single RDMA write, thereby minimizing the number of communicated packets.

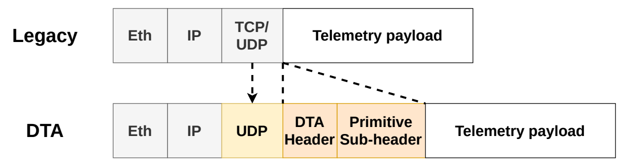


Figure 5: The structure of DTA report packets, compared with legacy TCP-based collection. DTA is designed to be transparent to the underlying network and monitoring system, and these headers will be entirely parsed and processed by the DTA network switches

8.5 DTA to Controller Interface/Socket

The DTA collector service is designed to allocate data structures and share essential metadata with the DTA translator during an initialization phase where the telemetry collection system is starting. The collector will then only spend CPU resources to answer incoming telemetry queries. This is achieved by presenting an interface to the controller or operator, where questions can be answered using the primitives that are configured for collection. All querying logic is abstracted away, and all telemetry communication between the controller and network will go through the DTA interface. Similarly to how TCP abstracts away congestion control, fragmentation, and so forth, are we abstracting away primitive processing, batching, indexing, among other low-level details while presenting a socket-like interface for efficient telemetry transfer to the control plane.

8.6 DTA Report Structure

We designed a simple packet structure to be used for telemetry data transfer between individual telemetry reporting switches, and DTA translators. Our goal is to allow telemetry data to be forwarded across legacy network, without requiring a complete network overhaul for supporting DTA-based telemetry collection. Therefore, we chose to encapsulate DTA operations within UDP, allowing for legacy network support and even the potential for NAT traversal on its way to collection. One might want to omit the UDP header in cases where pure IP-based forwarding is enough, thereby reducing the bandwidth overheads slightly. DTA is designed to allow ease-of-integration with current telemetry monitoring systems, and requires just a small additional header for specifying DTA primitive specifics as shown in Figure 5.

The *DTA header* is shared for all DTA primitives, and is used to specify the underlying DTA primitive that is requested by including small *Opcode* field. Alongside this, the base header also reserves 8 bits for specifying flags, which could be either global (i.e., for all primitives) or primitive-specific. This includes an *immediate flag*, which is planned to be used to alert the collector of important information that should trigger an interrupt for immediate processing of an incoming report [1] and an *essential flag* as discussed in section 8.7. The structure of the *Primitive sub-header* is dependant on the DTA primitive that is requested by the previous header. This header will, in the case of key-write packets, specify the telemetry key and a requested level of redundancy. The size of the telemetry key is dependant on the telemetry monitoring system. For example, scenarios where the monitoring system reports per-flow information will likely have the size of the telemetry key be equal to the size of the flow 5-tuple (i.e., 13 bytes). Telemetry collection designs where several different data categories are collected simultaneously requires different opcodes to specify the structure of the primitive sub-header, to ensure correct field lengths according to the telemetry requirements. The redundancy value is used as a representation of how aging-resistant this data should be, and is used as the base for selecting a level of redundancy (N) in the key-value primitive as discussed later in section **??**.

8.7 Reliable Collection of Essential Data

We propose the addition of a flag in DTA packet headers to state that telemetry information is *essential*, which signals that the reported information should be highly prioritized for collection in cases of congestion or system outage. A congested RDMA connection requires explicitly dropping telemetry reports during a resynchronization grace-period 7.3. DTA operations carrying the *essential* flag however would not be dropped at the translator. Instead, these are routed down to the translator CPU for temporary storage during collection downtime. After queue-pair resynchronization, when the grace period is over and RDMA connection is once again online, these packets can be re-inserted into the translator pipeline.

Such a design allows for telemetry collection *even when the collector is temporarily offline* of up to $10Gbps$, where the temporary collection bottleneck becomes the virtual link between the switch ASIC and switch CPU. However, this will never be a limiting factor in cases where the RDMA connection is online, and is used only as a fallback during potential collection failures and congestion events at the collector.

9. IMPLEMENTATION

The DTA switch pipelines are implemented in approximately $2.1K$ lines of code, split between the translator and a proof-of-concept DTA-coupled INT reporter. These contain full support of the Key-Write and Append primitives. This is accompanied by approximately 900 lines of Python to handle the switch controllers, including the RDMA-initiator script that is discussed in Section 9.2.1. DTA requires a low level of system logic at the actual collectors, due to an overwhelming majority of collector work being offloaded into the network, and required just a little over 700 lined of C++ for the RDMA collector service. Our entire codebase as of the time of writing has been released open source to the public[2].

[1]The DTA immediate flag is a proposal, and shall leverage the RDMA immediate functionality as specified by the Infiniband protocol to allow immediate CPU-handling at the collector.

[2]The full DTA codebase is available at <GIT LINK>

9.1 DTA Reporters

The reporter pipeline is written in P4_16 for Tofino, and is deployed on a BF2556X-1T switch. Controller functionality is written in Python, running on the switch CPU. The reporter controller is responsible for populating forwarding- and collector lookup-tables, containing destination IP addresses towards key-value storages and data lists. Full support of the DTA Key-Write and Append primitives is included in the proof-of-concept implementation, together with basic INT-based monitoring capacity for generating the telemetry information. DTA report packets are generated based on user traffic through packet mirroring, while normal user traffic remains unchanged. Original packets are forwarded as normal user traffic, while the clones are used as the base for crafting DTA traffic for telemetry collection. The Reporter simply places in the DTA packets what kind of DTA primitive to trigger, together with primitive-specific metadata. This can be for example to write an INT-carried network path under the flowID key using the Key-Write primitive.

The destination IP addresss of DTA reports are decided on a per-primitive basis. Key-Write traffic is implemented as a distributed key-write store, where a hash of the telemetry key is used as the base for deciding the destination collector that will store this information. This design choice allows for horizontal scaling of Key-Write collection capacity by deploying more servers, and updating the collector-mapping lookup tables hosted in each reporter. Append traffic selects a collector based on the chosen list ID, as decided by pre-loaded lookup tables. This ensures that all per-category telemetry data is efficiently aggregated in a single location, and telemetry scaling can be achieved manually by deploying additional lists to host data for the Append primitive. JL: Does this paragraph fit better in design?

9.2 DTA Translators

The translator pipeline is written in P4_16 for Tofino, and is deployed on a BF2556X-1T switch. Controller functionality is written in Python, running on the switch CPU. The translator controller is responsible for configuring the switch ASIC according to the DTA specification, including setting up multicast-rules to ensure correct multi-RDMA generation for the Key-Write, Key-Increment, and Sketch-Merge primitives, and configuring the address-calculating functions according to the memory state at the collector. The controller

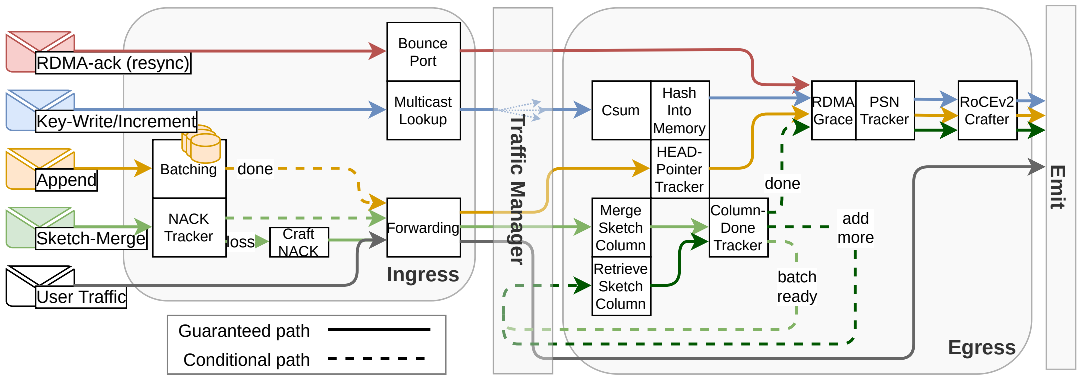


Figure 6: A translator pipeline with support for Key-Write, Key-Increment, Append, and Sketch-Merge. Five different paths exist for pipeline traversal, used to process different types of network traffic in parallel while efficiently sharing pipeline logic

is also responsible for initiating RDMA connections for each primitive, as described in Section 9.2.1.

Our current translator pipeline has full support for both the Key-Write and Append primitives, and is designed to efficiently re-use logic between these two and the potential addition of the Key-Increment and Sketch-Merge primitives. There are five main pipeline-traversing paths for different types of network traffic, as shown in Figure 6.

9.2.1 *Initializing RDMA Connections*

Tofino switches do not have native support for handling RDMA connections. Dedicating limited ASIC resources for setting up RDMA connections is incredibly wasteful, because these processes only occur once during system setup and they are not performance critical. We instead instruct the translator switch CPU to act as a virtual RDMA client during the RDMA initiation phase, sending fake RDMA connection traffic from the Tofino controller service for each. The ASIC will simply forward these to the correct RDMA network card, and send the RDMA replies down to the switch CPU. Initial RDMA traffic is parsed by the switch CPU, where essential metadata regarding the initialized RDMA queue-pair are parsed and inserted into the Tofino ASIC. This phase extracts all RDMA metadata that is required for pushing full control of the RDMA connection into the Tofino ASIC, where all subsequent RDMA communications are generated after the initiation phase is finished.

This initialization process is repeated for every storage structure at the collector. The metadata that is extracted for each RDMA queue-pair connection includes:

Queue-Pair Numbers The RDMA network card automatically assigns each connection a unique queue pair identifier, which is included in every RDMA packet from the translator to collector. Per-storage queue-pair numbers are written to an in-ASIC match-action table, and used when crafting RDMA-WRITE packets and processing RDMA acknowledgements from the collector.

Remote Keys Memory buffers that are registered with RDMA get assigned keys that function as passwords to prevent unauthorized remote memory calls. These keys are parsed during the connection phase, and are inserted into an in-ASIC match-action table for use when crafting RDMA-WRITE packets.

Initial Packet Sequence Numbers (PSNs) The RDMA network card gets assigned a random initial PSN, and advertises this value to the translator. The controller parses this value, and updates in-ASIC registers that track per-queue-pair PSN numbers.

Memory information Essential memory information is also extracted during this phase. These are the size of the allocated memory buffer, and the memory address to the buffer. These are both essential for crafting RDMA-WRITE calls for incoming DTA packets, and are written to in-ASIC match-action tables for the primitive that is able to write to that memory buffer.

GA: I would massively thin down this. here it is more a tech report.

9.2.2 *Key-Write & Key-Increment Implementation*

JL: This section is modified to also include the Key-Increment primitive. Please check that it generalizes to both of these

The Key-Write and Key-Increment primitives follow the same fundamental logic, with the main difference being the type of RDMA operation that they should trigger (see the difference between Figure 4a and 4b). Key-Write packets will trigger a WRITE operation, thereby overwriting the old data that occupied the memory slots, while Key-Increment will instead trigger Fetch-and-Add to increment the counters in the memory slots.

DTA Key-Write and Key-Increment packets shall generate N RoCEv2 WRITE or Fetch-and-Add packets respective. These N packets shall hold identical payloads, where the addresses point to N different memory locations where data redundancies are located. This requires us to inject N packets into the egress pipeline, allowing us to mutate these into N different RoCEv2 packets. This mutation is achieved through multicasting, where the switch controller pre-configures multicast-rules for each $< N, collector >$ tuple. The rules are configured to inject N packet clones into the egress pipeline for the port to that collector NIC, where the packets hold their own value $n \in 0, 1, ..., N - 1$ to be used for differentiating the redundant entries.

A lookup table has been pre-populated with metadata regarding the key-value store, including RDMA values, the memory address of the allocated storage buffer, the total size of the storage, and an index in the register that stores the queue-pair PSN tracker. The native CRC extern is used to calculate the key checksum that will be concatenated to the key-value entries, and to map $< key, n >$ into a slot-selecting hash which determines the destination memory addresses. This hash is further mapped into a range equal to the size of the key-value memory buffer through a match-action table, which maps the number of storage entries into a bitmask that can be used to perform modulo-operations by powers of 2[3]. The calculated memory slot is easily translated into a buffer-offset by multiplying it with the size of a single key-value slot (i.e., the sum of the checksum and data value lengths). This is followed by having the packet process the RDMA grace period enforcing block, to verify that RDMA traffic is not currently halted. Please see section **??** for a description of this block.

Finally, the N injected packets reach the RoCEv2-crafting block, which mutates these into the RoCEv2 standard. The switch then emits the N packets, which will trigger the N memory operations for updating the key-value store to include the new data.

JL: fixed hash-input endianness in Translator to speed up querying

9.2.3 *Append Implementation*

DTA Append packets insert telemetry data into one of several data lists. These lists are implemented as ring-buffers, where a pointer is kept in the translator to track where in the buffer to append the next piece of incoming data. A naive approach would be to simply generate one RDMA packet for each incoming operation, inserting these values one-by-one. However, values for each list will be written sequentially and contiguously in memory, which allows us to significantly reduce the number of required RDMA calls by batching multiple values together and writing them to memory with a single memory operation.

This is achieved by storing incoming Append-data in SRAM during ingress processing. Our current implementation supports batching of data on a per-list and per-ingress-pipe basis, with batches containing 16 elements each[4]. One might want to design batch-storage in a compressed way by using just a single register to store all 15 cached telemetry entries until a new batch is ready for insertion into collector memory, resulting in a very efficient use of memory logic. However, a design like this would require significant recirculation during RDMA-creation to read back this data, which would degrade the collection performance. Instead, we store Append-data in 15 different registers, each of size 255, to store per-list data that will be used as the base for crafting the next RDMA call. An initial register is used to track the number of data units that have been stored so far for each list, which is used either to decide which of the 15 registers to write into, or to signal that the stored data should be read back and RDMA-creation should be triggered. This design results in egress-traffic and RDMA-creation being triggered just for every $16th$ packet in each list, where these packets have a payload containing 16 units of Append-data, resulting in a significant reduction of RDMA traffic (and therefore improved performance)[5].

Egress contains the logic necessary for issuing an RDMA operation to append the Append operation information into the specified list. The packet starts by looking up primitive-specific metadata for the chosen list in a pre-populated match-action table, containing essential RDMA information, the size of the allocated buffer, and a register index that holds the head and PSN trackers. This is followed by processing the per-list head-pointer tracking register, which increments the stored value by the size of the batch, which will be used as an offset when calculating the destination memory address for the data [6].

Append traffic then enters the same RDMA-generating pipeline as Key-Write traffic, including RDMA grace enforcement, PSN tracking, and RoCEv2 crafting. The pipeline yields a single RDMA operation being sent for every 16 incoming per-list append packets.

9.2.4 *Sketch-Merge Implementation*

The DTA Sketch-Merge primitive is designed around highly pipelined architectures, and is therefore a good fit for implementation on the Tofino ASIC as shown in Figure 6. This primitive is not implemented in the current DTA translator pipeline, but the proposed design presented in this section adheres to the restrictions of the Tofino architecture and is designed to efficiently utilize the resources available to allow it to co-exist with other DTA primitives.

Sketch-Merge packets that enter the translator pipeline hold two different pieces of information: all counters in a specific

[3]This implementation choice effectively limits the number of key-value storage slots to powers of 2, and is a simple workaround to the lack of flexible modulo support in the switch architecture. An alternative could be to hard-code a fixed key-value storage size, which allows power-of-2 modulo through compile-time known bitshifting.

[4]The size of the allocated lists should be a multiple of the batch size, to allow resetting the head pointer when the end of the buffer has been reached, without leaving empty gaps in the allocated buffer. Our implementation currently forces both of these to be powers of 2, which fulfills this requirement.

[5]The current batch-design assumes a constant stream of Append-operations to each list. A sudden disruption of incoming data can lead to failure to complete a batch, which would mean that this data will not be written to storage until the telemetry stream continues. In cases where telemetry disruptions are expected, one might want to periodically flush idle batches into storage.

[6]Moving the head-tracking functionality after the Grace block would prevent data gaps in the list during times of collection congestion, if that is preferred.

column (e.g., 5 32-bit counters in the case of a standard 5-row sketch), and an integer declaring which column these counters belong to. NACK tracking can be implemented through a simple register containing counters for each network switch that can report sketches. Out-of-order reports will not be merged into the aggregation-sketch, and are instead mutated into a NACK that is sent back to the reporter to make it retransmit the lost report(s).

Valid sketch counters are merged into the local aggregation-sketch, where SALU logic performs the sketch-specific aggregation function (e.g., addition or max). A column-completeness register is used to detect fully populated sketch columns by having each updated column also increment the corresponding completeness-counter. A fully aggregated column is signalled when the counter value equals the total number of switches that report to this collector. The total number of completed columns is signalled through a single counter, where a threshold triggers column-merging down to the collector.

Tofino is limited to a single memory operator per register during a single pipeline traversal, which therefore requires recirculating the RDMA-building packet $B-1$ times to populate the RDMA payload with all B sketch columns.

A single RDMA packet can only write to contiguous memory, Sketch-Merge therefore writes the sketch to collector memory rotated, where the columns are written as rows as seen in Figure 4d. However, sketch-queries are performed through psuedo-random accesses, and this transformation should therefore not result in a performance degradation compared with traditional non-rotated representations.

9.2.5 *Queue-Pair Resynchronization*

9.3 Collector JL: rename. Collector-RDMA Interface? JL: Move/rewrite into design instead?

JL: we implemented socket in X lines of code... c++... basically does conversion between rdma and xx

The DTA collector service is written in C++, using standard Infiniband RDMA libraries and the RDMA Communication Manager (RDMA_CM) for setting up queue pair connections between the collector and translator. Server BIOS has been optimized for high-throughput RDMA, and all RDMA-registered memory is allocated on 1 GB huge tables. We recommend recompiling the Infiniband drivers to allow support for physical memory addresses, since random accesses in large memory spaces might result in a performance degradation due to IOTLB cache misses while translating from virtual to physical addresses.

Each collector service supports registering one key-value region, to be used for the Key-Write and potential Key-Increment primitives, and several parallel lists for the Append primitive. These are all advertised through RDMA_CM using unique ports, allowing the translator controller to request queue-pair connections to each region one-by-one. The collector RDMA service responds back to the translator with an RDMA SEND operation after an RDMA queue-pair connection is established between the collector and translator for one of the storages. This response contains essential metadata about the attached storage, including the total size of allocated memory, and the memory address to the start of the storage buffer.

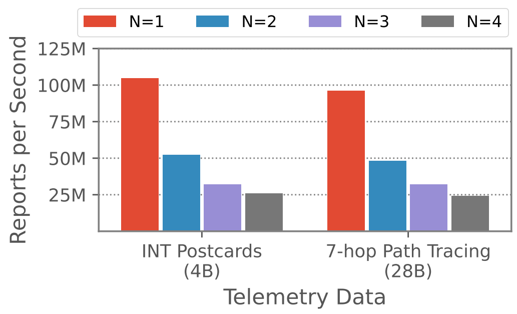


Figure 7: Single-NIC collection-rate of DTA Key-Write operations on a 4GiB key-value store, at various redundancy levels N and payload sizes. The RDMA payloads include a 4B concatenated checksum. Note how the telemetry data size does not impact performance until we reach line-rate at 32B payloads JL: Make payload sizes more "real" e.g., 28B is path tracing + queue depth or something

Initializing RDMA connections is an essential task for the collector CPU. After that is done, the CPU enters an idling phase, since the rest of the collection tasks are entirely CPU-bypassing. The CPU can now dedicate its entire processing capacity to handling telemetry queries by reading the memory buffers that are advertised as DTA storage.

10. EVALUATION

JL: Remove all discussions that are in DART paper and cite that one instead. Only leave novel

All benchmarks in this section are performed using *a single port* of the Mellanox Bluefield-2 2x100G DPU as acting RDMA-capable network card, and using a BF2556X-1T Tofino-1 switch as both reporter and translator. The collection speeds that are presented in this section are entirely dependent on the speed of the RDMA hardware, i.e., the RDMA message rate of the network card, which is the current bottleneck in our system. We therefore expect that even higher collection rates should be achievable if one were to use a more powerful network card in the future, and our current translation pipeline to be compatible as long as the network card supports the RoCEv2 protocol. The TReX traffic generator is used to inject DTA packets into the translator switch.

10.1 Key-Write Primitive Performance

This section will evaluate the raw report-collection performance of the Key-Write primitive, and the effectiveness of Key-Write internals.

10.1.1 *Key-Write Collection Rate*

We have benchmarked the performance of the DTA Key-Write primitive in terms of collected reports per second, as presented in Figure 7. Telemetry traffic is generated with sequential keys and data, at steadily increasing packet rates. The final throughput is the speed at when the system does not experience any packet loss in a several second interval. This test was repeated at different levels of redundancy (N), and we notice the expected linear relationship between the throughput and level of redundancy since each incoming report will generate N RDMA packets towards the collector. The collection rate was unaffected by increases in the telemetry data size, until the translator reached egress line-rate of $100Gbps$ at data sizes of $28B$.

10.1.2 Redundancy Effectiveness

JL: This section is heavily based on the hotnets paper The

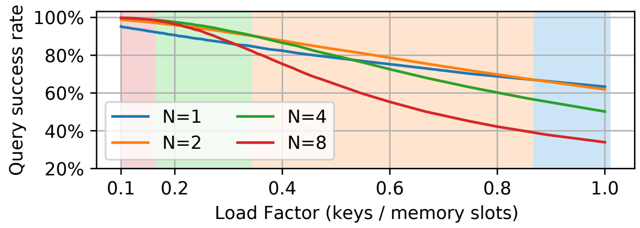


Figure 8: Average query success rates delivered by the Key-Write primitive, depending on the key-value store load factor and the number of addresses per key (N). The background color indicates optimal N in each interval.

probabilistic nature of the DTA Key-Write primitive cannot guarantee final queryability on a given reported key. We show in Figure 8 how the query success rate depends on the load factor (i.e., the total number of telemetry keys over available memory addresses), and the number of memory addresses that each key can write to. There is a clear efficiency improvement by having keys write to $N > 1$ memory addresses when the storage load factor is in reasonable intervals, and the background color in Figure 8 indicate which number of addresses per key (N) delivered the highest key queryability in each interval.

Recall that RDMA lacks the ability to write into multiple memory addresses with a single packet, and that DTA overcomes this by generating N distinct RDMA packets for writing the redundancy to memory. Requiring N DMA calls per DTA operation results in a decrease of collection throughput, as shown earlier in Figure 7. Determining an optimal redundancy level therefore has to be a balance between an enhanced data queryability and a reduction in collection performance. $N = 2$ appears to be a generally good compromise, showing great queryability improvements over $N = 1$.

10.1.3 Data Longevity

JL: Explain figure. Simulation, testing how long until data ages out

10.1.4 Query Answer Correctness

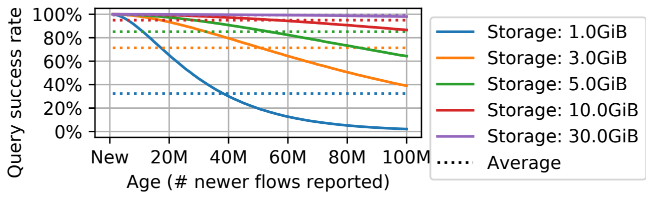


Figure 9: DTA Key-Write ages out stale data. This figure shows INT 5-hop path tracing queryability of 100 million flows at various storage sizes. The results are based on storing 5x32-bit values with 32-bit checksums, where each key writes into $N = 2$ different memory addresses. JL: make GB->GiB

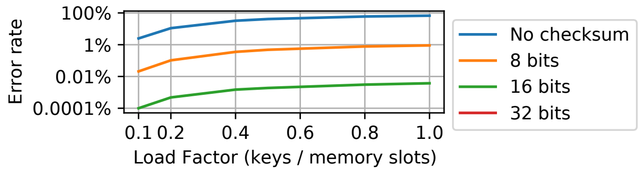


Figure 10: The probability of a query returning the wrong answer in error, due to address and checksum collisions.

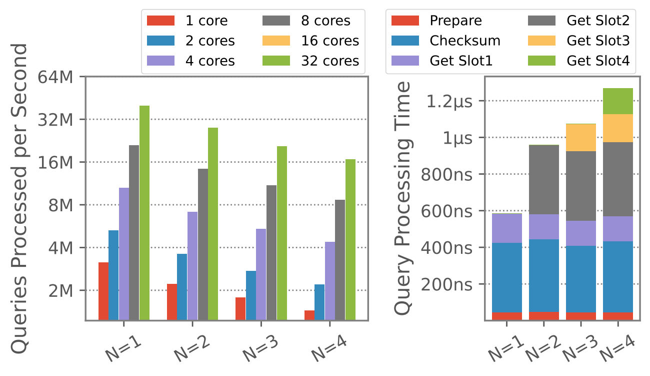
JL: Section is heavily based on hotnets paper

There is a theoretical risk of the Key-Write primitive returning incorrect query results if two or more different keys hash into the same memory address, while they at the same time also have the same concatenated checksum. JL: There is so much more than this. Can we fit some theory in the paper? Maybe as appendix? Figure 10 shows results after extensive tests, where multiple simulations of 100M keys have been performed at various storage sizes in an attempt to recreate the theoretically predicted incorrectness. These results clearly show the impact from having key-based checksums included in the DTA Key-Write data structure, with increased lengths greatly reducing the risk of errors. Our simulations with 32-bit key-checksums fail to reproduce these return-error cases, due to the incredibly low probability that these events occur, even while the data structure is under immense load.

Return error cases can be further reduced through data consensus during the query phase, where multiple identical redundancy slots are required to return back a query response. Requiring consensus will reduce the overall query success rates, and is only recommended in scenarios where incorrect query results can not be tolerated regardless of their frequency. See section **??** for a deeper discussion on Key-Write correctness.

10.1.5 Query Speed

Querying data that has been reported using the DTA Key-Write primitive requires calculating several hashes at the collector. The number of hash functions to calculate will be $N + 1$, one for each of the redundancy slots to locate them in memory, and an additional one to calculate the concatenated checksum that is used to validate the query answer. A query validation algorithm without consensus can however immediately answer a query when the first slot with a valid



(a) Query rate (b) Per-query breakdown

Figure 11: Querying performance of the DTA Key-Write primitive at different levels of redundancy, while querying INT postcards.

checksum was found, since a valid checksum would be the only criteria for validation. In this case, we would expect a reduction in the average query processing speeds since the processor would often not require iterating over all N redundancy slots to return back a query answer. However, here we evaluate the *worst case* performance, when the collector has to process every redundancy slot before a query answer is compiled.

Figure 11a shows the speed at which the collector can answer incoming telemetry queries through the Key-Write primitive. The overall query success probabilities correspond to the previously presented results from Figure 8. We instructed the collector to query a set of sequential telemetry keys ranging from 0 to $100M$ in-order, with a key-value data structure of size $4GiB$ containing $4B$ INT postcards with $4B$ concatenated checksums for query validation.

Figure **??** shows the cost at the collector CPU for retrieving per-key data from memory. We clearly see how the level of redundancy leads to an increased computational time for processing a query, which is further shown by a decrease in the rate at which queries can be answered in Figure 11a. However, query processing can easily be parallelized, where each thread processes a non-overlapping set of telemetry keys, and we see a near-linear increase in the query answering rate alongside the number of allocated query-processing cores in Figure 11a.

JL: draw conclusion. key-val is not expected to query everything, but it almost can. point back to recommending N=2, again mention this

10.2 Append Primitive Performance

JL: explain test in Figure 12

JL: outline test. explain that this is benchmarking the egress performance (since we lack traffic generators powerfull enough to saturate this). It is now more likely that Tofino ASIC Ingress would bottleneck in terms of link speeds or clock

JL: Explain very clearly and emphasize that the 100G LINK

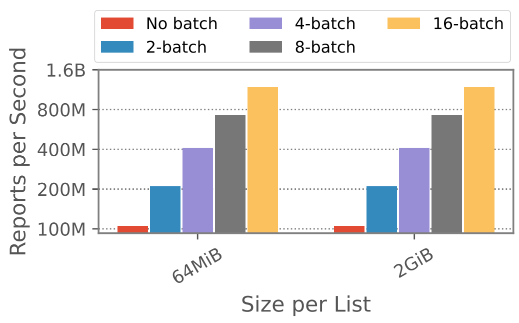


Figure 12: Single-NIC ingestion rate of DTA Append operations, at different batch sizes. The telemetry data entries in the lists are 32-bit integers. We see a linear performance increase until we achieve line-rate with batches of 4 or greater. Note that these are translator ingress-rates, while egress rates will be reduced by a factor of the batch size.

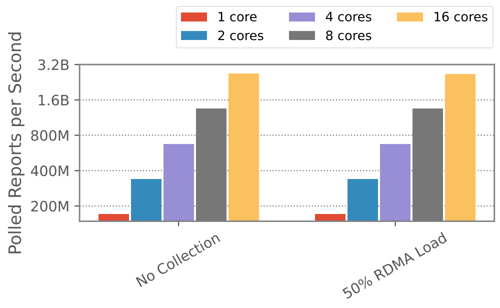


Figure 13: The speed at which appended data can be processed by the collector CPU. Note how the parallel RDMA traffic has a negligible impact on the data retrieval rate. JL: too similar to collection speed.JL: Add breakdown beside, to help explain how simple it it (why it's so fast and efficient)

is the bottleneck, and that we collect at line-rate. The next step would be to replace RoCEv2 to reduce header overheads.

JL: Memory retrieval is further improved by built-in caching (in actual HW architecture of server), due to memory being pulled sequentially from memory.

10.3 Append Querying Rate

JL: Key-Write at different redundancies and querying-threads

JL: Append at different number of lists (one thread querying per list). This eliminates tail-pointer race conditions

JL: Also benchmark in parallel with active collection? Show impact of collecting and querying simultaneously. Maybe run collection at 50% rate. Retrieving 10 billion data entries, benchmarked. translator is appending data to the same lists that are queried JL: Retrieves a value at the tail pointer, stores this aside in a new location, and increments the tail.

10.4 DTA Collection Capabilities

JL: Per-switch generation rates of monitoring systems come from their papers. State numbers and cite sources. Present in table format, together with brief description of mapping into DTA? (everything hypothetical)

JL: Marple: evaluating packet counters using properties of non-mergeable aggregations (figure 10 and end of section 5.2). Let's say the DTA epoch is shorter than cache eviction

JL: Possibly not in eval? Since this is potential performance (extrapolating from our primitive tests) and their paper numbers

$$required\_collector\_nics = \lceil \frac{num\_switches * switch\_report\_rate}{max\_primitive\_rate} \rceil$$

10.5 Queue-Pair Resynchronization

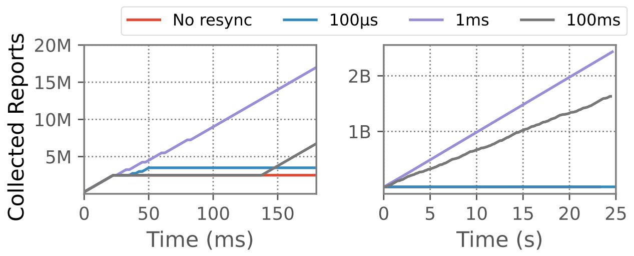


Figure 14: Translator-to-NIC queue-pair resynchronization, at varying grace period lengths. We see how DTA quickly and reliably manages to resynchronize the system after periods of collector-side congestion. Note how a too short grace period fails to reliably resynchronize the connection. The left figure is zooming in to the attempted resynchronization.JL: keep no resync as throughput (for motivation figure). Mention how we collect 99.xx% even when system is saturated as shown here

The RDMA connections have to be actively resynchronized when a packet loss has occurred. To evaluate the efficiency of our resynchronization mechanism, we sent a constant stream of DTA Key-Write traffic to the translator at a high enough rate, where we start to detect sporadic packet loss on the RDMA-NIC side of the system. In parallel to the Key-Write operations, we run a stream of Append traffic with incrementing values into a single list at a fixed inter-packet gap. We run this traffic through the system for a set period of time, and analyze the content of the Append-list. This list can now be seen as a representation of the NIC congestion, since periods of desynchronization and loss will result in gaps in the incrementing values contained in the list.

Figure 14 shows the effectiveness of our queue-pair resynchronization mechanism at various lengths of grace periods, compared with a system without active resynchronization. This figure motivates the need for including a grace period post-resynchronization, to allow the NIC to flush internal buffers of invalid RDMA traffic before accepting new telemetry data. Our tests show that a grace period of $1ms$ is enough to allow for queue-pair resynchronization, while grace periods of $100\mu s$ or shorter fails to keep the system stable post-resynchronization. Very long grace periods, on the order of $100ms$ or longer, manage to stabilize the system but are very wasteful due to unnecessarily long periods of disabled RDMA throughput.

Resource	Cost w/o batching	Cost w/ 16-batching
SRAM	5.4%	8.5% (+3.1%)
VLIW Instruction	6.5%	8.1% (+1.6%)
Match Crossbar	4.8%	14.4% (+9.6%)
Hash Bits	8.6%	16.0% (+7.4%)
Hash Dist Unit	12.5%	34.7% (+22.2%)
Meter ALU	8.3%	41.7% (+33.4%)

Table 3: Full resource costs of the translator, while supporting Key-Write and Append with size 16 batches for 255 lists of 32-bit values, compared with a pipeline that omits batch-building support for Append JL: include more batch sizes, and visualize

10.6 Translator Resource Footprint

DTA translators are responsible for several tailored processes, some of which have a high footprint on Tofino resources. Especially the batch-functionality for the Append primitive has a high resource cost in terms of stateful ALU logic (meter ALU), since a non-recirculated batch-reconstructing algorithm requires the ability to read all batch entries from memory during a single pipeline traversal. Table 3 shows the total resource usage of our current translator pipeline, both with and without built-in Append-batch support. It is clear that there is a significant resource cost for including Append-batching, which is expected due to the high amount of memory logic. However, batching also has the potential for a tenfold increase in collection throughput, and we therefore argue that this is still a worthwhile cost. A compromise is to reduce the size of these batches, the size of which linearly correlates with the number of additional meter ALU calls.

Building in support for more Append-lists does not require additional logic in the ASIC, it just necessitates more statefulness for keeping per-list information (e.g., head-pointers and per-list batched data). Note how the actual SRAM size requirement has just a 3.1% increase imposed by batching, which shows that the translator can support much more complex list setups than the 255 lists that are included at the time of evaluation.

We demonstrate here that DTA translators are already feasible in first-generation Tofino switches, even while including memory-intensive batch-building functionality. It is likely possible to design even more complex DTA primitives and aggregation functions than is presented here, given that the current-generation Tofino-2 has the capacity for significantly more memory logic than its predecessor.

10.7 Network Overheads

JL: Compare DTA vs RoCEv2 vs TCP in terms of network bandwidth. Maybe couple with Append performance. How much BW is 1Gpps?

11. LIMITATIONS

ToDo: *Explain limitations of current DTA design*

11.1 Limited Aggregation Capabilities

ToDo: *Explain how we are limited in data pre-processing and merge-capabilities*

JL: We can't really do RDMA Reads to efficiently merge data

JL: We can definitely utilize collector CPU to perform these, and likely significantly faster than current state-of-the-art.
Example: an Append list containing tuples <address,value,function>
CPU can just iterate over this list and perform this action.
Main drawback here is that it doesn't fit our paper story of zero-cpu

11.2 Real-time Reactivity

ToDo: *Discuss how CPU easily can immediately react to events depending on complex rules* JL: We can do this (e.g., RDMA Immediate or CPU constantly polling lists), but not necessarily "real-time" as in part of in-line processing

11.3 Efficiency of Append-Batching with Larger Data Slots

JL: Due to register busses max 32b output

INTEL REVIEWERS: we want to discuss how the register 32-bit bus width limits the data sizes that we can batch. Working around this limitation (e.g., splitting data across several register) is costly in terms of SALU calls. Conclusion being that we likely reduce the batch size (e.g., 16B slots can do batches of 4, instead of 16 as we can do with 4B data). This seems reasonable

12. DISCUSSION

ToDo: *Add a discussion. Feel free to modify how you see fit :)*

12.1 Non-Telemetry Usecases

JL: Database acceleration

12.2 Translation vs Programmable SmartNICs

ToDo: *Discuss the use of SmartNICs for DTA collection*

JL: Translation is cheaper than programmable NICs (legacy RDMA support). DTA COULD support a hybrid of translators and SmartNICs (maybe some primitives require SmartNIC collection, while others do not? No reason why DTA would not be future-proof here for DTA-optimized NICs). Translators work in the meantime

JL: Let's say opcode 0x01 is KeyWrite-1 (designed around translator), while 0x11 is KeyWrite-2 (designed around SmartNIC and Cuckoo). Both from user-perspective used for the same thing, but different implementations. JL: Maybe even the same opcode, just that the translator/smartnic chooses underlying algorithm to process the request? Plug-and-play :)

JL: SmartNICs would allow more complex central processing. Also (likely) easier to design around memory reads

JL: Vision for standardized DTA, allowing vendors to build in support in fixed-function NICs?

12.3 Read-based Writes

ToDo: *Discuss what we could do if we could read storage before writing. New collision mitigation or aggregation functions*

12.4 Large-scale Deployments

ToDo: *Discuss how DTA might be deployed in hyperscale networks.*
Requiring several collection clusters. Reporters already hash into one of several collectors.
Possibly segment network to reduce cross-network report tranmissions (one reporter to distant collector)

12.5 Collection Epochs and History

ToDo: *Discuss the need for epoch-based collection. DRAM is required for high-speed collection. Periodic transfer to disk is required for persistent historical storage.*

13. RELATED WORKS

ToDo: *Add related works*

13.1 TEA

ToDo: *Discuss TEA*

14. CONCLUSION

In this paper, we presented DTA, an RDMA-based approach to telemetry collection. DTA leverages the collector's ToR as a translator that both aggregates network-wide telemetry and is responsible for writing the data in a queryable form directly into memory. The benefit of our work is twofold, where we save CPU on both storing the data and its aggregation and is readily deployable with commodity RDMA NICs and programmable switches. We envision that similar principles can also aid many non-telemetry applications such as database acceleration [22] or ???.

Acknowledgements

Text goes here

15. REFERENCES

[1] R. B. Basat, X. Chen, G. Einziger, S. L. Feibish, D. Raz, and M. Yu. Routing oblivious measurement analytics. In *2020 IFIP Networking Conference (Networking)*, pages 449–457. IEEE, 2020.

[2] R. Ben Basat, S. Ramanathan, Y. Li, G. Antichi, M. Yu, and M. Mitzenmacher. Pint: probabilistic in-band network telemetry. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 662–680, 2020.

[3] M. Charikar, K. Chen, and M. Farach-Colton. Finding frequent items in data streams. *Theoretical Computer Science*, 312(1):3–15, 2004.

[4] X. Chen, S. Landau-Feibish, M. Braverman, and J. Rexford. Beaucoup: Answering many network traffic queries, one memory update at a time. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 226–239, 2020.

[5] G. Cormode and S. Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms*, 55(1):58–75, 2005.

[6] A. Dragojević, D. Narayanan, M. Castro, and O. Hodson. Farm: Fast remote memory. In *11th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 14)*, pages 401–414, 2014.

[7] D. Firestone, A. Putnam, S. Mundkur, D. Chiou, A. Dabagh, M. Andrewartha, H. Angepat, V. Bhanu, A. Caulfield, E. Chung, et al. Azure accelerated networking: Smartnics in the public cloud. In *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*, pages 51–66, 2018.

[8] P. Flajolet, É. Fusy, O. Gandouet, and F. Meunier. Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm. In *Discrete Mathematics and Theoretical Computer Science*, pages 137–156. Discrete Mathematics and Theoretical Computer Science, 2007.

[9] S. Grant, A. Yelam, M. Bland, and A. C. Snoeren. Smartnic performance isolation with fairnic: Programmable networking for the cloud. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 681–693, 2020.

[10] T. P. A. W. Group. Telemetry report format specification. `https://github.com/p4lang/p4-applications/blob/master/docs/telemetry_report_latest.pdf`. Accessed: 2021-06-23.

[11] C. Guo, L. Yuan, D. Xiang, Y. Dang, R. Huang, D. Maltz, Z. Liu, V. Wang, B. Pang, H. Chen, et al. Pingmesh: A large-scale system for data center network latency measurement and analysis. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, pages 139–152, 2015.

[12] A. Gupta, R. Harrison, M. Canini, N. Feamster, J. Rexford, and W. Willinger. Sonata: Query-driven streaming network telemetry. In *Proceedings of the 2018 conference of the ACM special interest group on data communication*, pages 357–371, 2018.

[13] Intel. In-band network telemetry detects network performance issues. `https://builders.intel.com/docs/networkbuilders/in-band-network-telemetry-detects-network-performance-issues.pdf`. Accessed: 2021-06-04.

[14] A. Kalia, M. Kaminsky, and D. G. Andersen. Design guidelines for high performance {RDMA} systems. In *2016 {USENIX} Annual Technical Conference ({USENIX}{ATC} 16)*, pages 437–450, 2016.

[15] C. Kim, A. Sivaraman, N. Katta, A. Bas, A. Dixit, and L. J. Wobker. In-band network telemetry via programmable dataplanes. In *ACM SIGCOMM*, 2015.

[16] J. Langlet, R. Ben-Basat, S. Ramanathan, G. Oliaro, M. Mitzenmacher, M. Yu, and G. Antichi. Zero-cpu collection with direct telemetry access. In *Proceedings of the Twentieth ACM Workshop on Hot Topics in Networks*, pages 108–115, 2021.

[17] Y. Li, R. Miao, C. Kim, and M. Yu. Flowradar: A better netflow for data centers. In *13th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 16)*, pages 311–324, 2016.

[18] S. Narayana, A. Sivaraman, V. Nathan, P. Goyal, V. Arun, M. Alizadeh, V. Jeyakumar, and C. Kim. Language-directed hardware design for network performance monitoring. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 85–98, 2017.

[19] J. Sonchack, A. J. Aviv, E. Keller, and J. M. Smith. Turboflow: Information rich flow record generation on commodity switches. In *Proceedings of the Thirteenth EuroSys Conference*, pages 1–16, 2018.

[20] P. Systems. Pensando dsc-100 distributed services card. `https://pensando.io/wp-content/uploads/2020/03/DSC-100-ProductBrief-v06.pdf`. Accessed: 2022-01-23.

[21] R. Teixeira, R. Harrison, A. Gupta, and J. Rexford. Packetscope: Monitoring the packet lifecycle inside a switch. In *Proceedings of the Symposium on SDN Research*, pages 76–82, 2020.

[22] M. Tirmazi, R. Ben Basat, J. Gao, and M. Yu. Cheetah: Accelerating database queries with switch pruning. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 2407–2422, 2020.

[23] Y. Zhao, K. Yang, Z. Liu, T. Yang, L. Chen, S. Liu, N. Zheng, R. Wang, H. Wu, Y. Wang, et al. Lightguardian: A full-visibility, lightweight, in-band telemetry system using sketchlets. In *NSDI*, pages 991–1010, 2021.

[24] Y. Zhou, C. Sun, H. H. Liu, R. Miao, S. Bai, B. Li, Z. Zheng, L. Zhu, Z. Shen, Y. Xi, et al. Flow event telemetry on programmable data plane. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 76–89, 2020.