Typhoon: An SDN Enhanced Real-Time Big Data Streaming Framework

Junguk Cho
University of Utah
junguk.cho@utah.edu

Hyunseok Chang
Nokia Bell Labs
hyunseok.chang@nokia-bell-labs.com

Sarit Mukherjee
Nokia Bell Labs
sarit.mukherjee@nokia-bell-labs.com

T.V. Lakshman
Nokia Bell Labs
tv.lakshman@nokia-bell-labs.com

Jacobus Van der Merwe
University of Utah
kobus@cs.utah.edu

ABSTRACT

Stream processing pipelines operated by current big data streaming frameworks present two problems. First, the pipelines are not flexible, controllable, and programmable enough to accommodate dynamic streaming application needs. Second, the application-level data routing over the pipelines do not exhibit optimal performance for increasingly common one-to-many communication. To address these problems, we propose an SDN-based real-time big data streaming framework called Typhoon, that tightly integrates SDN functionality into a real-time stream framework. By partially offloading application-layer data routing and control to the network layer via SDN interfaces and protocols, Typhoon provides on-the-fly programmability of both the application and network layers, and achieve high-performance data routing. In addition, Typhoon SDN controller exposes cross-layer information, from both the application and the network, to SDN control plane applications to extend the framework's functionality. We introduce several SDN control plane applications to illustrate these benefits.

CCS CONCEPTS

• **Computer systems organization** → *Distributed architectures*;
• **Networks** → *Cross-layer protocols*; *Programmable networks*;

KEYWORDS

Realtime Streaming Framework; SDN

ACM Reference format:
Junguk Cho, Hyunseok Chang, Sarit Mukherjee, T.V. Lakshman, and Jacobus Van der Merwe. 2017. Typhoon: An SDN Enhanced Real-Time Big Data Streaming Framework. In *Proceedings of CoNEXT '17, Incheon, Korea, December 12–15, 2017,* 13 pages.
https://doi.org/TBA

CoNEXT '17, December 12–15, 2017, Incheon, Korea

ACM ISBN TBA…$TBA
https://doi.org/TBA

1 INTRODUCTION

The rapid growth in big data has resulted in the emergence of versatile stream processing frameworks, enabling a variety of real-time stream processing applications such as business intelligence, event monitoring and detection, process control, data mining, machine learning, etc. To support the increasing number of stream processing use cases, existing open-source stream processing frameworks [3, 5, 6, 22, 43, 48, 55] provide desirable performance characteristics such as high throughput and low latency in stream processing, scalability, elasticity, as well as dependability properties such as availability and reliability [54].

Another important quality of a stream processing framework, especially for production environments, is *runtime flexibility*. In the context of stream processing, runtime flexibility implies the ability of a deployed stream application to dynamically adjust its processing pipeline and modify accompanying tasks in real-time. The need for such flexibility is particularly motivated by the following scenarios and trends.

Streaming pipeline growth: Stream pipelines continue to grow in scale and complexity (e.g., pipelines of 4K processing elements and clusters of 8K server nodes [32, 56]). This increases the likelihood that already deployed pipelines require some functional changes (e.g., improved anomaly detection algorithm, new business intelligence metrics) or iterative maintenance updates. With such expanded streaming pipelines, it is also more likely that some task will encounter faulty hardware [35, 50], and need to be migrated away without service degradation to meet SLA requirements [14].
IoT-driven data explosion: With a rapidly growing ecosystem of diverse IoT devices (e.g., 26 billion by 2020 [11]) and their tight integration with the cloud [27], re-purposing of an existing processing pipeline for different types of IoT-generated data in the data center may require hot-swapping different types of pre-processing, transformation or post-processing tasks (e.g., to handle varying data tuple schema) while the main pipeline is still active.
Non-determinism in data sources: When data streams are generated by high-entropy sources (e.g., tweets generated by human [15, 34], live video analytics [59]) or collected from geographically distributed wild environments [31], unpredictability and variability are common attributes of data. To support such non-determinism in input data, the processing pipeline must continue to adapt to changing data patterns (e.g., uniform vs. skewed workloads) and refine data sanitization logic without rebooting the pipeline.

Live debugging: Troubleshooting a live system is not easy, especially with a multitude of interacting tasks in a large pipeline [43]. At a minimum it requires a non-intrusive and interactive means to inspect pipeline state while handling real data flows in the system. For this, a selected portion of the pipeline may need to be instrumented on the fly (similar to dynamic instrumentation in programming [60]), or the intermediate pipeline may have to be tapped into or rerouted temporarily (similar to dynamic tracing in operating systems [45]).
Interactive data mining: One popular way to gain insight from streaming data is interactive data mining [36], where dynamically constructed queries are performed on existing streaming data pipelines. This requires additional query processing computations (e.g., combination of window, filter, join operations and custom mining algorithms) be dynamically plugged and unplugged at the main data pipelines.

All of these scenarios require that pipelined processing logic and internal data routing be dynamically modified at runtime. However, the support for this type of application-level flexibility has received relatively little attention in existing stream processing platforms. Typically they are designed from the ground up with the assumption that stream pipeline structures remain the same during their lifetime, which significantly simplifies the management of internal data routing. As a result, however, if the processing logic of an active stream application ever needs to be modified, either the application must go through a sequence of "shutdown, modification and restart" actions, or an extra instance of the (modified) application must be up and running for instant swapping [13]. Shutdown-restart actions can be time consuming for large-scale stream topologies, and can cause any in-flight or buffered tuple data to be lost. Neither of them is acceptable in real-time environments or mission-critical deployments (e.g., earthquake monitoring). Instant application swapping can be very resource inefficient as the infrastructure usage is doubled even for a slight application modification. Platform-supported ad-hoc rolling upgrade mechanisms often lead to performance degradation or other compatibility issues [17].

Flexibility-aside, the application-level data routing in existing stream frameworks does not exhibit optimal performance for certain types of communication patterns. In distributed computational systems such as stream processing, serialization is known as the main bottleneck for data object transfer (e.g., 60–90% of total transfer time [42]). The serialization overhead becomes particularly expensive when the data routing of a node requires *broadcasting* a data tuple to multiple next-hop destination nodes. In this case, the node must perform multiple serialization computations because each copy of the data tuple carries distinct metadata for each destination. As application-level data broadcast becomes increasingly common in modern streaming pipelines [40], they can suffer from significant performance degradation.

In this paper, we address these problems of application-level data routing (i.e., flexibility and performance) in existing stream processing frameworks. We design an SDN-enhanced streaming framework, called *Typhoon*. In designing Typhoon we started with the basic observation that a stream application relies on graph-based communication patterns among processing nodes, which are analogous to those in computer networks [44]. Our approach is based on *cross-layer design*, where application-layer data routing and delivery functions are partially offloaded to the network layer. This idea is inspired by the analogy between application-level data tuple routing and network-level packet steering, and the recent advances on the centralized control and programmability for the latter, driven by Software Defined Networking (SDN). To achieve centralized control on stream processing's data routing, the main challenges are: (1) the timely collection and centralized management of per-node routing state, and (2) the design of well-defined interface and protocol for seamlessly updating internal routing state without violating the application's data computing logic.

To address these challenges with Typhoon's SDN-based approach, we decouple the routing state from application nodes, and offload the management of the routing state to the network (i.e., SDN controller and switches), under the framework's coordination. In this model, the routing functions performed by individual nodes are flexibly updated by the network's SDN control plane, which carries necessary routing state into the application-level routing functions. During routing updates, the network and the framework work together to avoid any disruption to the ongoing data computations in a running topology. On the data plane, the network is programmed to deliver data tuples from one node to another based on the application-level routing decisions. If an application's routing decision requires broadcasting data tuples to multiple nodes, the underlying SDN data plane is programmed to replicate the tuples as necessary at the network layer, without introducing any extra serialization overhead at the application layer. The tight coupling between the framework and the network provides additional unique opportunities for the framework. Typhoon enables extending the useful capabilities of a stream processing framework, without modifying the framework itself, but simply by building and deploying SDN control plane applications. These applications leverage *cross-layer information* from the network (e.g., port/flow statistics and status events) and application (e.g., worker statistics) layers to make application-level decisions (e.g., topology scaling, fault recovery).

To the best of our knowledge, Typhoon is the first system that integrates SDN into a real-time streaming framework. We make the following specific contributions in this paper: (i) We design the SDN-based Typhoon framework which tightly integrates SDN functionality into a real-time stream framework to provide highly flexible stream pipeline and high-performance application-level data broadcast; (ii) We design several SDN control plane applications that can enhance the framework functionality by leveraging cross-layer information available from the framework and the network, and demonstrate their benefits; (iii) We implement a Typhoon prototype based on the Storm framework, and evaluate the prototype implementation.

2 BACKGROUND

In this section, we provide a brief overview of a stream processing framework without getting into the details of any particular system. As highlighted in Fig. 1, a generic stream framework consists of: (i) a *streaming manager* which manages and schedules the execution of stream applications submitted to the framework, (ii) a *cluster of compute hosts* with (iii) per-host *worker agents* which launch

streaming application upon request from the streaming manager, (iv) *workers* which perform application-specific computation and routing logic, and (v) a *central coordinator* which coordinates the communication among the components.

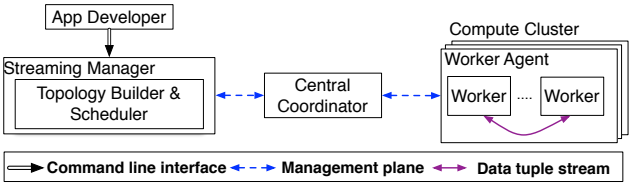


Figure 1: Stream framework architecture.

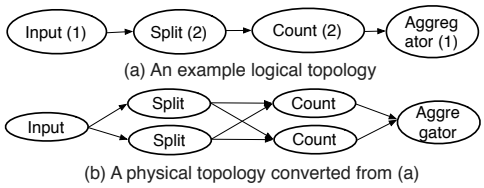


Figure 2: Example topology.

Stream processing topologies: When a stream application is submitted to a stream processing framework, the topology builder constructs a *logical topology* from it by analyzing its processing logic. Fig. 2(a) shows a simple logical topology that implements a word count example. The input is the stream of sentences, and the final output is the list of words with counts of their occurrences. A logical topology defines how input data tuples generated at the source(s) are transformed into final output tuples at the sink(s) through a directed acyclic graph (DAG). Each node in a DAG defines (i) a data computing function which converts incoming tuples to output tuples at the node, (ii) a routing policy which controls how computed tuples are routed to the next-hop node(s), and (iii) the degree of parallelism for the node. The logical topology is specified in the application source code using the framework-provided APIs, and thus determined at the application compile time. Given a logical topology, the scheduler converts it into a *physical topology* (Fig. 2(b)) by considering node parallelism, and assigns it on available compute hosts based on scheduling policies and current cluster availability. During that time, the scheduler assigns to each physical node a unique ID and transport channel information (IP address and TCP port).

Topology deployment and maintenance: Once the assignment of a physical topology is determined by the scheduler, individual nodes in the topology are deployed as *workers* on assigned compute hosts, which then interconnect with one another based on assigned TCP transport channel information. Worker deployment is handled by per-host *worker agents* which are responsible for fetching application binaries and launching scheduled workers on a host. During this process, the worker agents and the scheduler are coordinated by the *central coordinator*, which notifies the worker agents of any new worker assignment by the scheduler. Any worker failure is detected from periodic heartbeats sent by workers, and the scheduler re-schedules a failed worker upon detection.

Data tuple routing policies: A stream processing framework can support several types of routing policies for individual workers to meet different application requirements [9, 20, 47]. *Key-based routing* forwards data tuples such that tuples with the same key always go to the same next-hop worker. This routing is generally used for stateful workers (e.g., caching, streaming top N [9], data mining and machine learning [47]) due to its memory efficiency. However, it may cause load imbalance in case of skewed input distributions. *Shuffle routing* involves round-robin routing which offers load balancing functionality. It is generally used for stateless computations as it may lead to higher memory usage for stateful workers [47]. *Global routing* forwards data tuples to one specific worker, and is typically used for a sink worker to aggregate final results. *All routing* sends copies of the same data tuples to every connected next worker.

Per-worker routing policy is driven by the following routing state maintained in each worker: (i) a local routing table composed of a set of available next-hop worker IDs and their corresponding TCP connections, and (ii) routing policy type and policy-specific state (e.g., counter for round robin routing, key index for key-based routing). In many existing frameworks such as Storm and Heron, this per-worker routing state originates from the submitted application's source code.

Data tuple transfer: The format of egress data tuples consists of the raw output from a data computing function, prepended by its metadata which include source/destination node IDs, output length, and stream type. The metadata is used for parsing, forwarding and multiplexing in destination nodes. When data tuples are transferred from one worker to another, they are converted into byte arrays by serialization, and converted back to the tuple format by deserialization.

3 TYPHOON ARCHITECTURE

In this section, we present the Typhoon architecture. We start by clarifying its design goals, and providing an architectural overview. This is followed by a detailed description of the architecture.

3.1 Design Goals

In architecting an SDN-based stream processing framework, we identify several key design goals. First, the per-node routing state in a stream application (e.g., a set of next-hop node IDs, routing policy type, and policy-specific state) must be *re-configurable at runtime* in order to support dynamic routing policy update. During routing reconfiguration, we must also ensure that it does not interrupt ongoing data computations (e.g., due to data tuple loss) within affected nodes. Second, we aim to offload as much routing functions (e.g., broadcasting) as possible to the network layer to *minimize application-level overhead*. Third, given the tight coupling between the framework and the network, we aim to design a *unified management layer* within the framework that controls both the applications and the underlying network under the framework's coordination. Finally, the resulting framework should be *evolvable*, in the sense that additional management functionality can easily

be built on top of the unified management layer to enhance the framework.

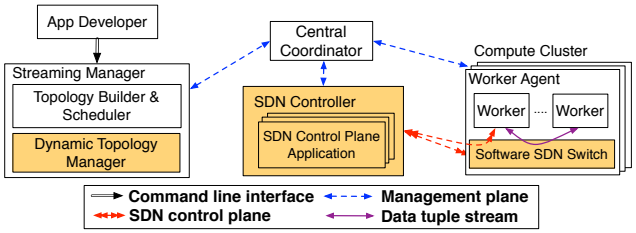


Figure 3: Typhoon architecture.

3.2 Architecture Overview & Workflow

Before presenting the detailed architectural design of Typhoon, we first describe its overall system workflow in an attempt to highlight how Typhoon differs from other existing systems described in Section 2.

As shown in Fig 3, Typhoon has the same basic components found in a generic streaming framework. The figure shows new components introduced by Typhoon in color. Specifically, the operating environment of Typhoon is distinct from that of existing stream frameworks in that hosts in the compute cluster contains a host-based *software SDN switch*. Workers that are deployed on a compute host are connected to the SDN switch running on the same host. Data tuple exchanges among deployed workers is enabled by SDN flow rules set up in these switches by the *SDN controller*. In other words, when a stream application is submitted to Typhoon, the application deployment procedure incorporates the configuration of these software switches. In addition to basic inter-worker communication, this SDN fabric enables the flexibility needs of Typhoon. A new *dynamic topology manager* module in the streaming manager is responsible for updating a running stream application upon request from an application user. Finally, the Typhoon framework enables *SDN control plane applications* to be deployed to realize a range of cross layer framework functions.

Given this extended streaming framework, the following provides a step-by-step procedure for deploying a new application in Typhoon: (i) **Topology build and schedule**: A logical topology is built from the submitted application, and then converted into a physical topology. While the scheduler assigns the physical topology to an available cluster, it determines, for each physical node, a compute host to deploy it, a unique worker ID, and its dedicated SDN switch port on the host. (ii) **Notification**: The SDN controller and all worker agents which are assigned part of the physical topology are notified by the central coordinator about the assignment. (iii) **Network setup:** The SDN controller sets up SDN flow rules into the SDN switches to interconnect assigned compute hosts and workers' SDN ports according to the physical topology. (iv) **Application setup:** The worker agent fetches application binaries, launches assigned workers, and attach them to the SDN switch. (v) **Data tuple communication:** Each worker is initialized and starts processing data tuples communicated via the host-based SDN switches.

In addition to basic stream processing, the Typhoon architecture enables the following *streaming application and pipeline reconfiguration*:

- **Per-node parallelism:** change the number of concurrent workers for a particular node in a logical topology.
- **Computation logic:** launch new workers with new computation logic in an existing topology.
- **Routing policy:** change routing type (e.g., from key-based to round robin), or change policy-specific parameters for routing (e.g., key indices for key-based routing).

To support such reconfigurations, the dynamic topology manager module updates the information of a running stream application in the coordinator when necessary. The following outlines the workflow steps involved with application and pipeline reconfiguration: (i) **Reconfiguration request**: Typhoon receives a reconfiguration request with reconfiguration options for an active application. (ii) **Topology reschedule**: The topology manager updates the corresponding logical topology. The scheduler then re-schedules it as necessary, and updates physical topology information in the coordinator. (iii) **Notification**: The SDN controller and affected worker agents are notified by the coordinator about the reconfiguration. (iv) **Network/application reconfiguration**: The SDN controller adds, deletes or updates flow rules in the SDN switches. The notified worker agents launch or kill workers based on updated physical topology information.

Table 1 summarizes global states maintained by the coordinator, and how different components are coordinated via it based on these states. In the rest of the section, we focus on the design of workers and the SDN controller, both of which are unique to Typhoon.

3.3 Worker Design

In Typhoon, workers are integrated with both the application framework and the SDN network, and thus its design plays a key role in Typhoon's overall operations. As shown in Fig. 4, the internal worker design is functionally divided into three layers. The *application computation layer* implements user-defined computation logic for incoming tuples, and thus remains unchanged in Typhoon. The *framework layer* provides necessary functions to support stream applications such as implementing routing policies, tuple formatting and de/serialization. The *I/O layer* handles conversion between data tuples from the framework and network packets from an SDN switch. In the following, we elaborate the design of the latter two layers.

3.3.1 Typhoon I/O layer. Interposing between the framework and the SDN network, the I/O layer is responsible for converting data tuples to network packets, and vice versa. One key design decision we make for worker-to-worker tuple transport is that Typhoon workers leverage a custom transport protocol instead of relying on application-level TCP connections. The need for the custom packetization I/O layer is motivated by this decision. An end-to-end transport protocol like TCP is not suitable for Typhoon, where next hop destinations of data tuples are determined by stateless SDN rule matching at the network layer, without relying on any end-to-end TCP state pre-established among workers. In addition, TCP-based worker communication is not optimal, in terms

States	Information	Writers	Readers
Logical topology	Topology ID, reconfiguration options, inter-node connectivity, node parallelism, per-node routing info	Streaming manager, SDN controller	Streaming manager, SDN controller
Physical topology	Topology ID, location of application binaries, per-worker assignment info (worker ID, hostname, SDN switch port)	Streaming manager	SDN controller, worker agents, workers
Worker agents	Hostname, used/available switch ports	Worker agents	Streaming manager, SDN controller

Table 1: Global states in Typhoon.

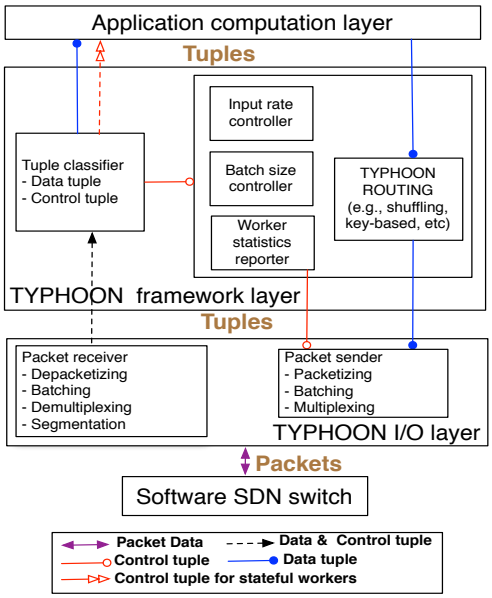


Figure 4: Typhoon worker design.

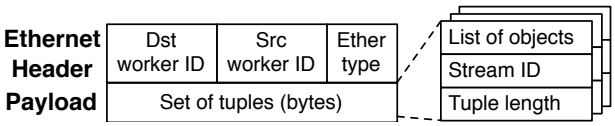


Figure 5: Typhoon packet and tuple format.

of throughput and latency, for one-to-many tuple communication, which is increasingly popular due to the collective communication pattern of HPC applications or publish-subscribe type of processing. One-to-many communication patterns require multiple serialization computations for each data tuple, which can lead to significant computation overhead and drop in throughput [42]. Besides, since a source worker sends tuples serially to each destination worker one at a time, the maximum latency experienced by destinations can grow proportional to the number of designation workers [40].

While avoiding TCP connections in the worker-level for these reasons, Typhoon leverages *host-level* TCP tunnels which interconnect different compute hosts where workers are deployed. These tunnels are used to reliably carry data tuples exchanged across hosts over the network, and to hide Typhoon's custom transport protocol format from the underlying physical network. Even with fixed inter-host TCP tunnels, data tuples can still be flexibly routed to different workers, as programmed by SDN flow rules.

For custom transport packets that carry data tuples, we adopt an Ethernet packet format with a custom EtherType as shown in Fig. 5 (see Section 3.4 for more detail on the custom EtherType). For SDN-based data forwarding, the Ethernet source/destination addresses are filled with source/destination worker IDs combined with application ID as an address prefix. The payload is populated by a set of tuples based on their size. This format is optimized for one-to-many communication; a source worker emits only one packet with its destination address field set to the broadcast address, and intermediate SDN switches can forward it to as many destinations as needed without multiple serializations since packet payload is identical for all destination workers.

Functionality-wise, the I/O layer realizes de/packetizing, de/-multiplexing, segmentation, and batching for correctness and performance as shown in Fig. 4. When a tuple is sent out, a pair of sender/destination worker IDs are determined by routing logic in the framework layer, and then inserted into the source/destination address fields in the Ethernet header. During packetization, more than one tuples with the same source/destination can be multiplexed into a single packet to save on throughput. When packets are received from the SDN switch, the I/O layer handles segmentation and demultiplexing of the packets based on source and destination worker IDs contained in the Ethernet header. To support different application requirements in terms of latency and throughput, the I/O layer is designed to support a configurable amount of batching when sending data tuples and packets to the framework and the SDN switch. The batch size can be flexibly configured based on the relative priority of latency and throughput on a per-application basis.

3.3.2 Typhoon framework layer. The framework layer plays a key role in the Typhoon architecture as it performs application-level routing policies, and directly interacts with the SDN control plane to reconfigure individual workers' routing states. As with traditional framework functionality, this layer also handles de/serialization of tuple objects sent to, or received from the application computation layer. In order to realize SDN-driven worker re-configuration, we leverage the SDN controller's ability to inject packets into the SDN data plane (e.g., `PacketOut` in OpenFlow [24]), but under the Typhoon unified tuple communication model. To this end, the framework layer is designed to handle special *control tuples* which are generated and injected by the SDN controller.

While having the same tuple format as data tuples, control tuples have streamID and tuple payload fields set differently (e.g.,

dedicated streamID, and re-configuration information in the payload) to distinguish them from data tuples. Incoming data tuples are always handed over to the upper application layer after deserialization, while incoming control tuples can be either consumed within the framework layer or passed to the upper application (Fig. 4), depending on their role.

```
/* Round robin based routing policy */
index = (counter++) % numNextHops;
dstWorker = nextHops[index];

/* Key-based routing policy */
hashedTuple = new Tuple(fieldA, fieldB);
hashVal = hash(hashedTuple);
index = hashVal % numNextHops;
dstWorker = nextHops[index];
```

Listing 1: Example data routing policies.

Let's examine how control tuples are used for different purposes in Typhoon. We first consider how a control tuple can update the internal routing state of a worker (e.g., increase or decrease the number of concurrent next-hop workers, or change routing policy type). To understand the procedure, let's examine what kind of routing state is maintained in available routing functions. Listing 1 illustrates how commonly used routing functions like round-robin and key-based routing are implemented in a stream framework. As shown above, any routing function, regardless of its routing policy, requires *nextHops* which is an array of next hop workers, and *numNextHops* which is the number of next hop workers. These are part of policy-independent routing state. There is also policy-specific routing state. For example, in round-robin based policy, the *counter* variable, which is incremented after every routing decision, is the state needed to pick the next destination. In key-based routing, a set of fields to hash on (e.g., *fieldA* and *fieldB*) is kept as policy-specific routing state.

Depending on which routing state needs to be changed, Typhoon's control tuples carry necessary information in their payload field. For example, in case of adjusting the number of next-hop workers, the updated values of *numNextHops* and *nextHops* are carried in a control tuple. If it needs to change a set of fields for key-based routing without changing the number of next-hop workers, a new set of fields are retrieved from a control tuple. The control tuples which carry an updated routing state are forwarded to the framework layer and the routing state is updated accordingly.

In addition to managing a worker's internal routing state, control tuples can also be used to support the framework functionality. For example, while the SDN controller collects network-level statistics for deployed workers by querying SDN switch, it can also collect application-level statistics (e.g., the number tuples sent or received) by injecting worker statistics requests into the workers via control tuples. Control tuples can also be used to change the processing rate in workers, or to adjust batch size for the I/O layer. In these framework-supporting roles, control tuples are consumed within the framework layer. Table 2 shows the supported control tuples in Typhoon. All control tuples except for `METRIC_RESP` are sent by the Typhoon SDN controller to workers.

The worker-level configurations in Typhoon provide not only run-time flexibility not available in existing frameworks, but also *finer-grained* controls for stream applications than existing frameworks which generally allow only topology-level configurations.

Control tuple type	Description
`ROUTING`	Update application routing information
`SIGNAL`	Flush a cache in workers
`METRIC_REQ`	Request worker's internal statistics
`METRIC_RESP`	Response to a metric request (e.g., queue status, number of emitted tuples)
`INPUT_RATE`	Control an input processing rate in workers
`ACTIVATE/DEACTIVATE`	(De)activate a topology by (un)throttling the first workers in a topology
`BATCH_SIZE`	Adjust tuple batch size

Table 2: Control tuples in Typhoon.

3.4 SDN Controller

As another key component of Typhoon, the SDN controller functions as a *unified management layer* which controls not only the SDN network, but also stream applications and the framework layer, all via the well-defined OpenFlow protocol interface [24]. The SDN controller directly controls data tuple transport among workers by programming SDN switches with `FlowMod` OpenFlow messages. At the same time, as shown in Section 3.3.2, it controls stream applications and the framework layer indirectly by leveraging control tuples carried in `PacketOut` OpenFlow messages.

For better manageability, the Typhoon SDN controller is designed as a *stateless* component, not maintaining any state about stream application deployments and available compute cluster. Instead, the SDN controller learns the framework's global states through the central coordinator as shown in Table 1, and generates flow rules based on the global states.

Table 3 shows the flow rules that are installed for data/control tuples in Typhoon. For data tuple communication among workers, the SDN controller installs flow rules (in Step 3 in Fig. 3), based on worker IDs and their corresponding switch ports assigned to a stream application. For transport packets that carry data tuples, Typhoon uses an Ethernet format with a custom EtherType (e.g., 0xffff), so that any unnecessary wildcards for unused IPv4 header can be avoided in rule processing of SDN switches. For remote tuple communication between workers running on different hosts, a separate tunneling port is designated to send and receive tuples via a TCP tunnel. When one worker wants to send the same tuple to multiple workers, a corresponding flow rule matches the broadcast address in the destination address field, and specifies those workers' ports in the output action.

As for control tuples that are handled by the framework layer to reconfigure workers' state, they are generated by the SDN controller, and sent to the SDN switch via `PacketOut` OpenFlow messages, and eventually delivered to connected workers by flow rules that match those tuples. In case the SDN controller wants to collect application-layer statistics from the framework layer, it sends a worker statistics request via a `PacketOut` message, and collects the requested statistics through a `PacketIn` message sent from a requested worker.

3.5 Stable Topology Update

While an active stream application can be re-configured based on updating SDN flow rules discussed in Section 3.4, reconfiguration without careful consideration can negatively affect the performance, as well as the accuracy and consistency of final results. For example, if workers send data tuples to newly added downstream workers

Tuple type	Worker communication	SDN flow rules	
Data tuple	Local transfer	match	**in_port**=[src worker's port], **dl_src**=[src worker ID], **dl_dst**=[dst worker ID], **ether_type**=[0xffff]
		action	**output**=[dst worker's port]
	Remote transfer (sender)	match	**in_port**=[src worker's port], **dl_src**=[src worker ID], **dl_dst**=[dst worker ID], **ether_type**=[0xffff]
		action	**set_tun_dst**=[peer's IP addr], **output**=[tunneling port]
	Remote transfer (receiver)	match	**in_port**=[tunneling port], **dl_src**=[src worker ID], **dl_dst**=[dst worker's ID]
		action	**output**=[dst worker's port]
	One-to-many transfer	match	**in_port**=[src worker's port], **dl_dst**=[BROADCAST], **ether_type**=[0xffff]
		action	**output**=[all dst workers' ports]
Control tuple	SDN controller to workers	match	**in_port**=[OFPP_CONTROLLER], **dl_dst**=[BROADCAST], **ether_type**=[0xffff]
		action	**output**=[all reconfigured dst workers' ports]
	Worker to SDN controller	match	**in_port**=[src worker's port], **dl_dst**=[OFPP_CONTROLLER], **ether_type**=[0xffff]
		action	**output**=[OFPP_CONTROLLER]

Table 3: SDN flow rules in Typhoon.

Worker type	Stateful	Stateless
In-memory cache	Yes	No
Routing policy	Key-based routing	Shuffling, global, all routing

Table 4: Type of workers.

which are not ready to receive tuples, or workers that were just killed as part of reconfiguration, those tuples will be lost. If an application must ensure the exactly-once processing for every tuple for reliability, lost tuples need to be detected and recovered [18, 46], which will hurt throughput performance. Even if guaranteed processing is not required, tuple loss can still introduce inaccuracy in application processing (e.g., top-N ranks). In another scenario, if a worker uses key-based routing policy, and the number of next-hop workers is changed due to scale up/down reconfiguration, key-based routing can no longer guarantee that tuples with the same key will go to the same worker as *numNextHops* in Listing 1 changes. If a worker accumulates data from incoming tuples using in-memory cache (e.g., time-based window operation), such internal worker state will be lost if the worker is killed off as part of a scale-down request.

To avoid these problems, we classify workers based on whether or not reconfiguration can potentially break consistency, and apply different topology update procedures based on worker type. Table 4 shows two types of workers and their common implementation characteristics. In the following, we describe detailed procedures for adding or removing a worker according to its type.

Stateless worker: When a stateless worker requires reconfiguration, Typhoon proceeds as follows to guarantee no tuple loss during reconfiguration (see Fig. 6(a)). If a reconfiguration requires adding a new worker (to increase node parallelism or to add new computation logic), a new worker is first launched, and SDN flow rules are set up between its predecessor worker(s) and the new worker. Finally, the predecessors' routing states are updated via control tuples. If an existing worker needs to be removed as part of reconfiguration, first its predecessors' routing states are updated via control tuples, so that no more data tuple is sent to the worker to be deleted. Then, once the worker finishes emitting any ongoing tuples, it is removed from the topology. The SDN flow rules interconnecting the worker and its predecessors are automatically removed due to idle timeout of the rule entries.

Stateful worker: When modifying stateful workers, we leverage the insights from common routing patterns of stateful workers in stream applications [9] to guarantee consistency. Stateful workers are generally used for time-windowing, and Listing 2 illustrates how they are typically implemented.

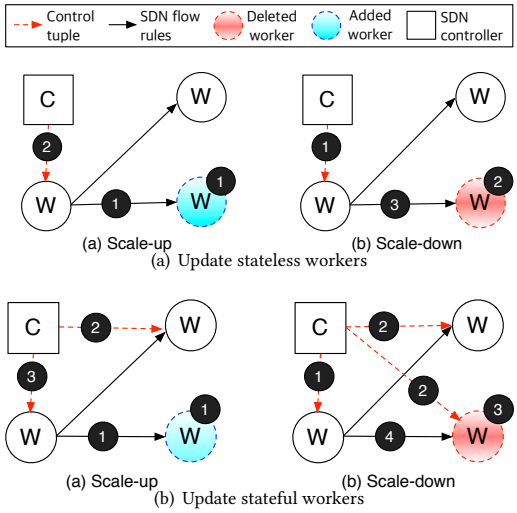


Figure 6: Stable topology update.

```
public class StatefulWorker {
  /* in-memory cache that stores currently processed results */
  Map<String, Integer> counts = new HashMap<String, Integer>();
  . . . .
  public Tuple execute(Tuple input) {
    if (TupleUtils.isSignalTuple(input))
      emitRankings(counts);      /* flush the cache */
    else
      addTupleInCounts(tuple);  /* add it to the cache */
  }
}
```

Listing 2: In-memory and key-based routing pattern.

As shown above, a stateful worker maintains currently processed results in a memory cache. Upon receiving a special *signal tuple*, it flushes the cache, and emits the current results to next-hop workers. In conventional streaming frameworks, signal tuples are generated and injected to support stream applications that want to perform scheduled processing. For stateful workers, signal tuples are commonly utilized to flush internal states along with key-based routing. In Typhoon, we leverage this pattern to guarantee consistency while updating stateful workers. Fig. 6(b) shows the sequence of updating stateful workers. The procedure is similar to the stateless worker case, except that the SDN controller injects signal tuples into the stateful worker to flush stored data after the first step, and right before reconfiguration.

4 SDN CONTROL PLANE APPLICATIONS

As described in Sections 3.4 and 3.5, the basic functionality of the SDN controller is to set up network-layer flows for data tuple communication among workers, and reconfigure the workers and their tuple communication flows. Besides this basic functionality, the SDN controller can exploit *cross-layer information* from the network (e.g., port/flow statistics and status events) and application (e.g., worker statistics) layers to enhance the framework functionality. In the following, we introduce control plane applications that one can build on top of the Typhoon SDN controller.

Fault detector: Since a stream application runs indefinitely as multiple workers deployed across hosts, failure handling is important. To detect worker failure, traditional frameworks generally rely on periodic heartbeats from workers. However, handling periodic heartbeats on a large-scale deployment is expensive [32], and delayed failure detection from heartbeat timeouts can cause non-negligible data loss. Instead, the Typhoon SDN controller detects a dead worker from an unexpected port removal event, and takes a proactive approach to update affected flow rules immediately, well before the dead worker is re-scheduled with heartbeat timeouts.

Live debugger: Debugging a deployed stream application (e.g., inspecting a particular worker's input/output tuples) is extremely useful. One way to support debugging in traditional stream frameworks is to provision special-purpose workers at application deployment time, which then receive and display copies of data tuples from a streaming pipeline [19]. This approach is not only inflexible (in terms of monitoring granularity, display format, and provisioning time), but can also degrade the application throughput performance due to additional application-level serializations. Instead, the Typhoon SDN controller can easily support highly flexible and efficient live debugging capability by dynamically adding a debug worker anywhere in a running topology and inserting packet-mirroring rules for selected tuples. The debug worker can be flexibly re-deployed with custom filtering logic and display format to meet application-specific debugging requirements.

Load balancer: Stream applications commonly use round-robin based shuffle routing to evenly distribute the workload of a particular computation logic among multiple workers. However, round-robin based load balancing can be unfair or can introduce straggling workers if the tuple size distribution is highly skewed or the underlying compute cluster is heterogeneous in terms of compute power. To overcome these problems, the Typhoon SDN controller can apply *SDN-level load balancing*, in which application-level routing decisions are fully offloaded to SDN. In this approach, a worker populates destination IDs for outgoing tuples randomly, instead of applying any routing, and the SDN switch rewrites their destination IDs in a weighted round robin fashion (e.g., using *select*-type `Group` in OpenFlow [24]) among multiple destinations, and forwards them accordingly. The weight associated with each destination can be dynamically adjusted by the SDN controller based on application-level (e.g., node's CPU load) and network-level (e.g., port statistics) information.

Auto scaler: The ability to auto-scale the number of concurrent workers in response to dynamically changing data volumes is critical in stream processing. In this case, network-level statistics collected from SDN switches are not sufficient to determine whether or not running workers are overloaded. Instead, the auto-scaler app leverages application-layer metrics (e.g., tuple queue level and tuple processing time) retrieved from ZooKeeper or workers, and initiates scale up/down operations via control tuples when the metrics reach predefined maximum and minimum thresholds.

5 PROTOTYPE IMPLEMENTATION

We implemented the Typhoon architecture using Apache Storm [7] as our base, and using Open vSwitch (OVS) and the Floodlight controller [16] to realize the SDN components. We choose Apache Storm as a baseline stream framework because it already provides common runtime facilities (e.g., centralized job scheduler and coordinator, per-host supervisor daemon), programming APIs for developing and deploying stream applications, and pluggable/extensible interfaces for key components (e.g., scheduler, network transport, coordinator), which allows us to re-use, replace or extend them relatively easily. Our implementation consists of 2000 lines of Java code and 800 lines of C code for implementing the OVS-integrated Typhoon data plane, and 4500 lines of Java code for extending Storm and integrating Typhoon with Floodlight control plane. Below we highlight key implementation details.

Central coordinator: Following Storm, Typhoon utilizes Apache ZooKeeper [38] as the central coordinator, and manages logical/-physical topologies as language-agnostic Thrift objects [53] in ZooKeeper. We extend the Thrift object definitions designed for Storm to include reconfigurable parameters and SDN configurations for logical and physical topologies, so that the SDN controller and other Typhoon components can be coordinated via ZooKeeper as shown in Table 1.

Typhoon streaming manager: We implement the Typhoon streaming manager by refactoring Nimbus in Storm, which performs central job management tasks such as building and scheduling topologies. In the streaming manager, we modify the topology builder component of Nimbus and implement the dynamic topology manager module. Together, they update necessary Typhoon global states in ZooKeeper during job submissions and reconfigurations. We also implement a custom Typhoon topology scheduler by leveraging Storm's pluggable scheduler interface (`IScheduler`). Replacing Storm's default round-robin scheduler, the Typhoon scheduler assigns topologically neighboring workers to the same compute node to minimize remote inter-worker communication.

Typhoon worker I/O layer: In order to implement the worker's I/O layer as explained in Section 3.3.1, we leverage the pluggable network transport interface (`IContext` and `IConnection`) in Storm. We implement the I/O layer as a *custom transport library* plugged into that interface, which replaces the `Netty`-based Storm's default transport implementation. Transferring data tuples as custom Ethernet packets between the framework layer and the SDN switch, the Typhoon's transport library must be able to create and process raw Ethernet packets in user space without introducing performance bottlenecks. To meet this requirement, we implement the library using the DPDK framework [10], and integrate it with DPDK-based userspace OVS as an SDN switch. One caveat is that while the Storm's network transport APIs are defined in Java, the DPDK framework only provides the C interface. Thus, we implement the transport library in two components; one written in Java, and the

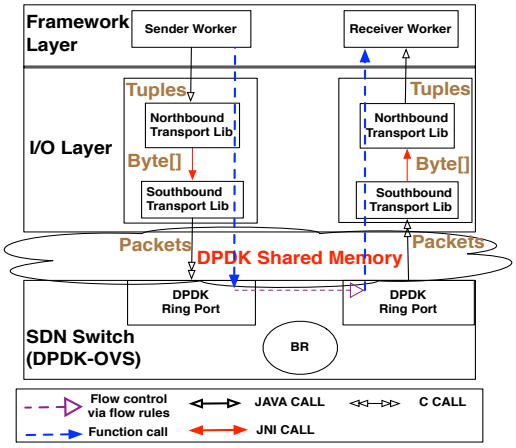


Figure 7: Typhoon data plane implementation.

other in C, and interconnect them via Java Native Interface (JNI). We refer to the framework-integrated Java component as a *northbound* transport library, and the network-integrated C component as a *southbound* transport library.

The southbound transport library transforms byte arrays of tuples received from the northbound library via JNI calls, into custom Ethernet packets to be sent to DPDK OVS via shared memory ring buffers. Conversely, it de-packetizes incoming Ethernet packets into tuple byte arrays. It sends and receives multiple tuples or packets in a batch to minimize the overhead of JNI function calls and DPDK-based inter-process communication. The workflow of the southbound transport is implemented as follows. (i) *egress workflow:* It receives a byte array of outgoing tuples with their source/destination worker IDs set via the northbound library's JNI call, and converts them into a set of Ethernet packets with the worker IDs in Ethernet headers and additional metadata (tuple length, packet segmentation info, etc.) prepended in payloads. Multiple small tuples with the same source/destination IDs are packed into one packet, while one large tuple is segmented into multiple packets. Finally, it sends out the packets via shared memory TX ring buffers of a target switch port. (ii) *ingress workflow:* It polls for incoming packets in shared memory RX ring buffers of DPDK OVS. Upon receiving packets, it de-packetizes them into a byte array consisting of a set of <source worker ID, payload length of a packet, payload>, and sends the constructed byte array to the northbound library via a JNI function call. (iii) *management workflow:* It can dynamically create or destroy shared memory ring buffers to connect or disconnect workers in the SDN switch. It can also adjust the batch size for existing workers for latency/throughput tradeoff.

The northbound transport library transforms tuple objects from the worker's framework layer into serialized byte format, and vice versa. It also exploits configurable batching to minimize JNI call overhead. The workflow of the northbound transport is implemented as follows. (i) *egress workflow*: It receives a list of tuple objects from the framework layer, and queues them up internally. When the number of queued objects reaches a batch size, it transforms the tuple objects into a byte array consisting of a set of <tuple length, tuple data> with source and destination worker IDs, and sends the byte array to the southbound library via JNI. (ii) *ingress workflow*: It receives a byte array of tuples with their length and source worker ID information from the southbound library via JNI, and converts them into a list of tuple objects. When multiple tuples are received as a single packet, it segments the payload in the byte array into multiple tuple objects. It forwards the constructed tuple objects to the framework layer.

Typhoon SDN control plane: We implement the Typhoon's SDN control plane using Java-based Floodlight. We re-use the tuple libraries from Storm to implement the control tuple communication in Floodlight, and use Apache Curator [2] for the interaction between Floodlight and ZooKeeper. We implement those control plane applications described in Section 4 and deploy them on Floodlight. Some of these applications interact with framework users via REST APIs, so that the users can leverage a Typhoon-provided framework service (e.g., topology reconfiguration and debugging services).

6 EVALUATION

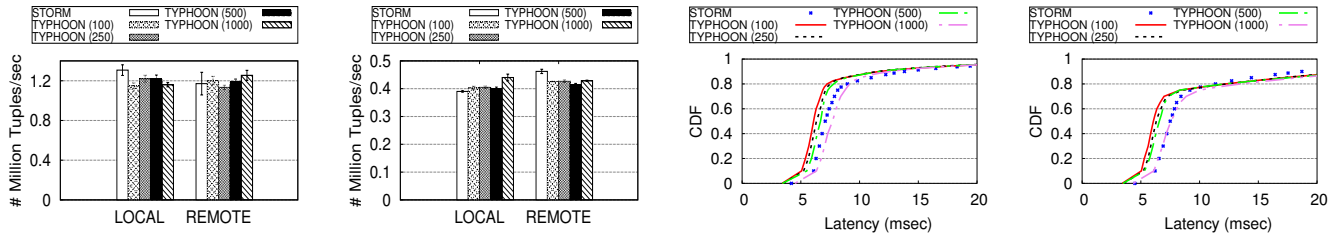
In this section, we evaluate the baseline performance of Typhoon and demonstrate its benefits as compared to Apache Storm. We deploy the Typhoon prototype and Storm on baremetal servers from the Emulab testbed [57], each with 32 cores, 64GB of memory and 10GB NICs. For Typhoon, one server is designated to run the streaming manager, Zookeeper and the SDN controller, while the rest servers run workers. In all experiments, we use Storm's default configurations with a round-robin topology scheduler for fair comparisons.

6.1 Baseline Performance

While Typhoon provides enhanced features (e.g., runtime reconfigurability, extensible SDN functionality), such capability should not come at the cost of degraded performance. Here we evaluate the performance of Typhoon's custom transport implementation in terms of throughput and latency, and compare it against Storm.

Tuple forwarding: We first evaluate the performance of data tuple forwarding in a simple topology consisting of two workers. A source worker injects a sequence of string tuples at maximum speed, and a sink worker checks the sequence numbers in the tuples. Fig. 8(a) shows the maximum throughput performance (tuples/sec) of this topology. LOCAL means both workers run on the same server, while REMOTE implies they are deployed on two servers. The number in the label indicates different batch sizes in Typhoon I/O. Typhoon and Storm show similar throughput in both cases, meaning that Typhoon's flexible I/O layer is still fast enough to handle maximum speed inputs from the application layer. The batch size has minimal effect in this experiment as the source worker emits tuples at maximum speed.

Tuple forwarding with reliability guarantee: To support stream applications with reliability requirements, Storm provides *guaranteed processing* [18], which ensures that each input tuple to a topology is fully processed (i.e., at least once) through the entire topology despite any worker/host failures. This is implemented by leveraging an application-level ACK mechanism, where special *acker* workers receive ACK messages for every tuple processing downstream to detect failed tuples, and notify input workers. If any



(a) Tuple forwarding. (b) Tuple forwarding with ACK. (c) Tuple latency (local). (d) Tuple latency (remote).

Figure 8: Storm vs. Typhoon performance comparison.

input tuple is not fully processed, it is replayed from input workers. Typhoon supports Storm's guaranteed processing by installing SDN flow rules for ackers when they are enabled. Fig. 8(b) compares the throughput performance of Storm and Typhoon when one acker worker is enabled for the above two-worker topology. Both system shows similar throughput, while the performance drops in half compared to Fig. 8(a) due to the extra computation and network overhead of the acker.

We also measure the end-to-end latency of tuple processing and plot their CDFs in Figs. 8(c) and 8(d). The latency is measured in the source worker which is notified from the acker worker when the processing of each tuple is completed. Latency becomes smaller as the batch size decreases in Typhoon I/O layer, which is expected. When the batch size is smaller than 500, Typhoon shows lower latency than Storm. We expect that the end-to-end latency without acker will show similar patterns with varying batch sizes.

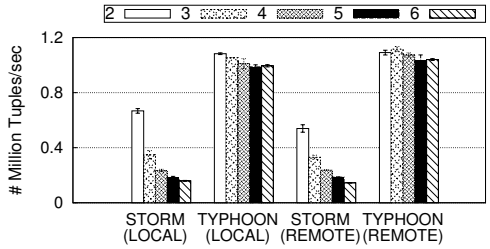
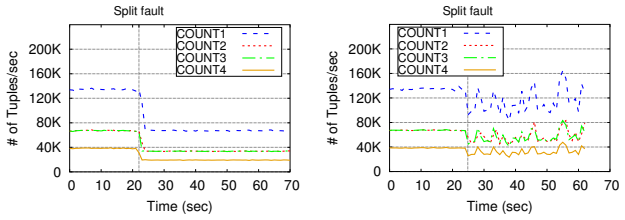


Figure 9: One-to-many communication.

One-to-many tuple forwarding: If a stream application leverages one-to-many routing, it can benefit from serialization-free network-level broadcasting in Typhoon. To demonstrate this benefit, we deploy a topology consisting of one source worker and a varying number of sink workers. The source worker broadcasts input tuples to all connected sink workers. Fig. 9 shows the throughput performance of Storm and Typhoon when the number of sink workers increases from two to six. The figure clearly shows the increasing performance gap between Storm and Typhoon. While the throughput of the former significantly drops with more sink workers due to multiple serializations, data copies and TCP overhead, the latter shows similar throughput regardless of the number of sink workers due to negligible packet copy overhead in OVS.

6.2 SDN Control Plane Applications

One key selling point of Typhoon is its extensible functionality via SDN control plane. We describe the implementation of several SDN control plane applications and demonstrate their usecases. In all experiments, we set the batch size to 100.



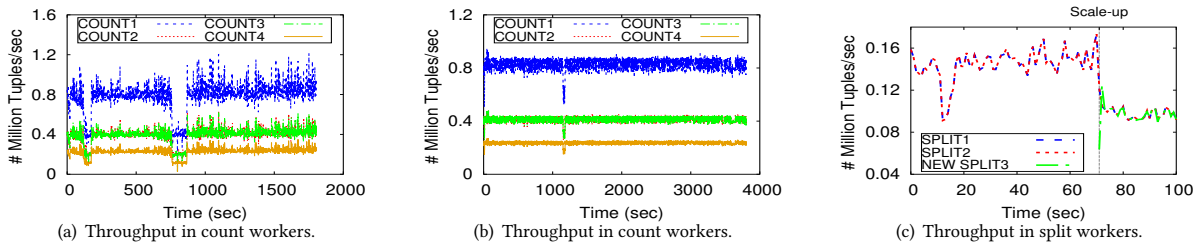
(a) Sink workers in Storm. (b) Sink workers in Typhoon.

Figure 10: Storm and Typhoon fault evaluation.

Fault detector: The fault detector SDN app is useful for fast fault detection and recovery in case running workers die. We use a word count topology shown in Fig. 2, with one source, two split workers and four count workers deployed on three servers. Shuffle and key-based routing are used between the source and split workers, and between split and count workers, respectively. Each count worker may receive a different amount of workload from preceding split workers if the key distribution is skewed. For evaluation, we intentionally cause `NullPointerException` in one split worker, and compare the fault recovery in Storm and Typhoon in Fig. 10.

In Storm, when a worker dies, it is locally detected and the worker gets restarted on the same server. Since it continuously fails, and is unable to send heartbeats (for 30 sec. in Storm by default), Nimbus re-assigns it on another server. During this time, count workers cannot receive any tuple from the faulty worker, and thus their throughput drops in half as shown in Fig. 10(a). Even after worker rescheduling, the throughput of counter workers remains in half since Nimbus does not realize the re-deployed worker is also faulty. Typhoon follows the same recovery mechanism as Storm, except that the fault detector receives `SwitchPortChanged` notification due to the worker fault, and immediately re-directs incoming tuples to the other alive split worker. As shown in Figs. 10(b) and 10(a), the average aggregate throughput of sink workers in Typhoon remains the same after sink worker failure, while in Storm, their aggregate throughput drops in half. The fluctuation in throughput in Typhoon is because the other split worker needs to process double the amount of tuples.

Auto-scaler: Next, we demonstrate the worker-level reconfiguration capability of Typhoon, in particular, auto-scaling workers. To demonstrate the impact of auto scaling, we deploy the same topology used for the fault detection scenario, but with a very high



(a) Throughput in count workers. (b) Throughput in count workers. (c) Throughput in split workers.

Figure 11: Typhoon auto scale-up performance.

input rate. Fig. 11(a) shows the throughput of four count workers in Storm. Occasional throughput drops are caused by the failure of their preceding split worker due to `OutOfMemoryError`. While the throughput is back up after the failed split worker is restarted, it is not a permanent solution. Instead, as shown in Fig. 11(c), the Typhoon's auto scaler app detects overloaded split workers, and initiates scale-up operation by introducing the third split worker, and re-distributes input tuples among three split workers. As a result, the throughput in count workers is much more stable afterwards as demonstrated in Fig. 11(b). While the figure shows a temporary dip in throughput, we verify that there was no packet drop or tuple loss during the Typhoon experiment.

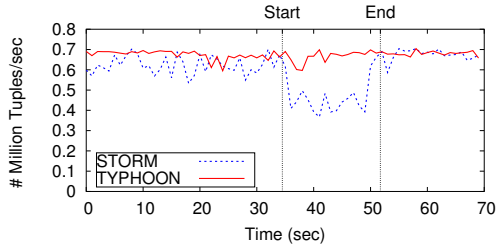


Figure 12: Live debugging overhead.

Live debugger: We compare the live debug system in Storm and Typhoon. In this experiment, we deploy a topology consisting of a source, a sink, and a debug worker for live logging. In both systems, live logging is temporarily activated from 18 sec., during which tuples from the source are replicated to the debug worker. As shown in Fig. 12, the throughput of the topology drops significantly in Storm due to serialization overheads, while Typhoon's throughput is not affected thanks to lightweight network-level packet copy. Table 5 summarizes the comparison of Storm and Typhoon for their live debugging capability.

Property	Debugging granularity	Resource requirement	Dynamic provisioning	Multiple serialization
Storm	An entire topology, or a set of workers	Pre-provisioned memory and TCP connections	No (predefined via `storm.yaml` or API in apps	Yes
Typhoon	Each worker	Memory allocated on demand	Yes	No

Table 5: Storm vs. Typhoon: live debugger comparison.

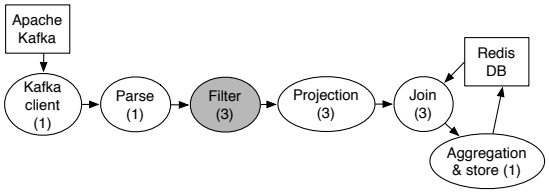


Figure 13: Yahoo advertisement analytics application.

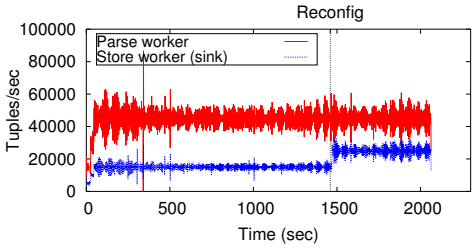


Figure 14: Runtime update on computation logic.

Computation logic reconfiguration: Next, we demonstrate Typhoon's capability to replace the computation logic of an active worker by using the Yahoo streaming benchmarks [30]. Simulating an advertisement analytics pipeline, the benchmark application performs six distinct computations in its pipeline, with Kafka as an input source and Redis as a database for join and aggregation workers (Fig. 13). The number in each computation indicates the number of workers. Except for the join and aggregation computations which maintain a local cache and a 10-second tuple window, the rest of computations are stateless. For reconfiguration, we focus on the filter worker which processes three different types of event tuples (e.g, view, click and purchase). In the initial deployment, the filter worker allows only view events, but let's assume we need to change the filtering logic such that it allows both view and click events. Typhoon enables computation logic change via Typhoon SDN controller. A user simply submits a reconfiguration request to the Typhoon SDN controller with updated topology information. Coordinated via ZooKeeper, Typhoon then deploys three new filter workers, connects them to parse and protection workers, and kills the old filter workers. The reconfiguration procedure does not require shut-down or topology hot swapping operations which are expensive. Fig. 14 confirms that windowed count increases after replacing filter workers as the new filtering logic allows more events.

7 RELATED WORK

Several open-source stream frameworks provide runtime reconfigurability, but not to the extent of Typhoon. For example, Storm [23] provides a rebalancing mechanism by which compute resources assigned to a physical topology can be adjusted, but the topology itself is fixed during its life time. Gearpump [4] allows dynamic reconfiguration at a logical topology level, but cannot switch routing policies at runtime [12]. Apex [1] supports dynamic modification of logical DAGs, but any possible modification must be pre-planned and implemented at the application design stage [8, 21]. In terms of research efforts, TimeStream [49] proposes a resilient substitution abstraction to replace any sub-graph of a streaming topology at runtime, but it does not allow dynamic routing policy change or composition of multiple routing policies. Load-aware routing policies for stream processing [47, 58] are point solutions focusing on data load balancing.

Early works on distributed stream processing [25, 29, 51] support adaptive re-partitions to address load imbalances on the fly. The load of Flux data flow operators [51] is monitored and dynamically repartitioned by a centralized controller. Borealis [25], based on its predecessor Aurora [29], has control lines which carry information to update operator's behavior (e.g., revised operator's parameters and functions). While there are similarities between Typhoon and their components, Flux does not support elastic scaling of parallel operators, and Borealis supports only limited elasticity using *box splitting* approaches adopted from Aurora. None of these systems supports dynamic user-defined computation logic update.

An SDN-based intra-host communication model [44] is in similar spirit as Typhoon, but no detailed architecture or prototype implementation is available. Authors of [40] focus on data tuple broadcasting and propose several broadcasting algorithms that reduce inter process communication overhead by using shared memory.

8 DISCUSSION

Cross-layer design. Typhoon presents a case of cross-layer design, leveraging tight integration between the network and streaming applications via SDN interfaces and programmability. While cross-layer designs are often met with caution and skepticism in fear of unintended consequences of cross-layer interactions and relatively poor sustainability [41], we argue for Typhoon's cross-layer approach as follows. First, the recent SDN innovation has introduced "SDN network hypervisors" [26, 28, 52], which empower data center tenants to operate their own fully isolated virtual SDN slices. This capability offers opportunities for data center tenant applications like streaming frameworks to easily interact with their own SDN slices, independently of data center network infrastructure, and while avoiding any potential conflicts with other cross-layer applications operated by other tenants [33]. Also, while we admit that any limitation of existing streaming frameworks could be addressed at the application layer (e.g., application-level proxy for flexible tuple routing), the ultimate benefit of Typhoon design comes from its extensible architecture beyond run-time flexibility (Section 4).

Stateful worker management. Our current design for reconfiguration supports limited types of stateful workers based on window operations and in-memory cache, as described in Section 3.5. However, given the emerging trends toward decoupling state from processing logic to an external storage [37, 39], we believe that Typhoon can be extended to support more general stateful workers by leveraging fine-grained run-time management of workers via control tuples (Table 2) and dynamic flow modification via the SDN controller. For example, in case of relocating a stateful worker from one host to another, Typhoon can simply "pause-and-resume" the worker via control tuples (e.g., `SIGNAL` and `(DE)ACTIVATE` tuples), while its state remains in an external storage.

Packet loss in software SDN switches. In typical Typhoon deployments, the main bottleneck will be the workers, not the SDN switches, and Typhoon can scale workers to avoid bottlenecks (e.g., auto-scaler in Section 6.2). However, tuple loss may still occur at the SDN switches (e.g., due to temporary TX/RX queue overflow) if workers send via Typhoon I/O layer a large batch of tuples to the switches at once. Such switch-level tuple drops need to be handled by an application-level ACK mechanism. To avoid switch-level packet drops and minimize expensive application-level recovery, one can apply moderate batch sizes in the worker I/O layer, and also use large sizes of TX/RX queues and packet buffers in the switches.

Impact of host-level TCP tunnels. For reliable tuple delivery, Typhoon maintains TCP tunnels at the host level, not worker level. This raises a question on its impact on data center wide network load balancing, since the host-level TCP tunnels will create more elephant flows, and thus can prevent ECMP from performing properly. This point is certainly true if the Typhoon framework and its applications are deployed in a dedicated compute cluster. However, if Typhoon is deployed as a regular tenant application running in virtual compute resources (e.g., VMs) interconnected with an encapsulated overlay (e.g., VxLAN, Geneve, STT), Typhoon's TCP tunnels do not reduce the entropy of data center's traffic distributions beyond what those overlays have. Even if Typhoon is deployed in a dedicated compute cluster, Typhoon's run-time flexibility allows any deployed worker to be relocated to a different host at run time. Thus, it can re-schedule any deployed worker pair to the same host if they generate significant remote communication traffic. More detailed analysis of performance implication is left for future work.

9 CONCLUSION

We present Typhoon, an SDN-based real-time stream framework that tightly integrates SDN functionality into a real-time stream framework to enhance its capabilities. Typhoon uses an SDN controller to control both network and applications for stream processing, using well-defined interfaces. Moreover, the Typhoon controller exposes cross-layer information to SDN control plane applications to manage active stream applications. We showcase several useful SDN control plane applications and present a thorough evaluation of the Typhoon prototype implementation. Compared to an existing stream framework, our results show better performance in terms of throughput and better managements of running stream applications.

Acknowledgements: We would like to thank our shepherd Monia Ghobadi and our reviewers for their feedback on earlier versions of this paper. We also thank Limin Wang for his technical help. This work was initiated when the primary author was an intern at Nokia Bell Labs, and is supported in part by the National Science Foundation under grant numbers 1302688 and 1305384.

REFERENCES

[1] Apache Apex. https://apex.apache.org.
[2] Apache Curator. http://curator.apache.org.
[3] Apache Flink. https://flink.apache.org.
[4] Apache Gearpump. http://gearpump.apache.org.
[5] Apache Samza. http://samza.apache.org.
[6] Apache Spark. http://spark.apache.org.
[7] Apache Storm. http://storm.apache.org.
[8] Apex Application Developer Guide. https://github.com/DataTorrent/docs/blob/master/docs/application_development.md.
[9] Common Topology Patterns. http://storm.apache.org/releases/1.0.3/Common-patterns.html.
[10] Data Plane Development Kit. http://dpdk.org.
[11] Gartner Says the Internet of Things Installed Base Will Grow to 26 Billion Units By 2020. https://www.gartner.com/newsroom/id/2636073.
[12] Gearpump - dynamic DAG. http://mail-archives.apache.org/mod_mbox/incubator-gearpump-user/201609.mbox/browser. Gearpump User Mailing List.
[13] How Spotify Scales Apache Storm. https://labs.spotify.com/2015/01/05/how-spotify-scales-apache-storm/.
[14] Microsft Azure SLA for Stream Analytics. https://azure.microsoft.com/support/legal/sla/stream-analytics/.
[15] New Tweets per second record, and how! https://blog.twitter.com/engineering/en_us/a/2013/new-tweets-per-second-record-and-how.html.
[16] Project Floodlight. http://www.projectfloodlight.org/floodlight/.
[17] STORM-634: Storm serialization changed to thrift to support rolling upgrade. https://github.com/apache/storm/pull/414.
[18] Storm Guaranteeing Message Processing. http://storm.apache.org/releases/current/Guaranteeing-message-processing.html.
[19] Storm topology event inspector. http://storm.apache.org/releases/2.0.0-SNAPSHOT/Eventlogging.html.
[20] Stream groupings. http://storm.apache.org/releases/2.0.0-SNAPSHOT/Concepts.html.
[21] Support for Dynamic Topology. http://mail-archives.apache.org/mod_mbox/apex-users/201608.mbox/browser. Apex Users Mailing List.
[22] Tigon. http://tigon.io.
[23] Understanding the Parallelism of a Storm Topology. http://storm.apache.org/releases/2.0.0-SNAPSHOT/Understanding-the-parallelism-of-a-Storm-topology.html.
[24] OpenFlow Switch Specification. https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-switch-v1.5.0.noipr.pdf, 2014.
[25] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina, et al. The Design of the Borealis Stream Processing Engine. In *Proc. CIDR*, 2005.
[26] A. Al-Shabibi et al. OpenVirteX: Make Your Virtual SDNs Programmable. In *Proc. HotSDN*, 2014.
[27] A. Botta, W. de Donato, V. Persico, and A. Pescapè. On the Integration of Cloud Computing and Internet of Things. In *Proc. IEEE International Conference on Future Internet of Things and Cloud*, 2014.
[28] Z. Bozakov and P. Papadimitriou. AutoSlice: Automated and Scalable Slicing for Software-Defined Networks. In *Proc. ACM CoNEXT*, 2012.
[29] M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Cetintemel, Y. Xing, and S. B. Zdonik. Scalable Distributed Stream Processing. In *Proc. CIDR*, 2003.
[30] S. Chintapalli et al. Benchmarking Streaming Computation Engines: Storm, Flink and Spark Streaming. In *Proc. IEEE International Parallel and Distributed Processing Symposium Workshops*, 2016.
[31] L. G. D. Estrin and M. S. G. Pottie. Instrumenting the World with Wireless Sensor Networks. In *Proc. IEEE ICASSP*, 2001.
[32] R. Evans. From Gust to Tempest: Scaling Storm. In *Proc. Hadoop Summit*, 2015.
[33] A. D. Ferguson, A. Guha, C. Liang, R. Fonseca, and S. Krishnamurthi. Participatory Networking: An API for Application Control of SDNs. In *Proc. ACM SIGCOMM*, 2013.
[34] A. Floratou, A. Agrawal, B. Graham, S. Rao, and K. Ramasamy. Dhalion: Self-Regulating Stream Processing in Heron. *VLDB Endowment*, 10(12), 2017.
[35] D. Ford, F. Labelle, F. I. Popovici, M. Stokely, and V.-A. Truong. Availability in Globally Distributed Storage Systems. In *Proc. USENIX OSDI*, 2010.
[36] A. Ghoting and S. Parthasarathy. Facilitating Interactive Distributed Data Stream Processing and Mining. In *Proc. IEEE International Parallel and Distributed Processing Symposium*, 2004.
[37] S. Hendrickson, S. Sturdevant, T. Harter, V. Venkataramani, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Serverless Computation with OpenLambda. In *Proc. 8th USENIX Workshop on HotCloud*, 2016.
[38] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *Proc. USENIX ATC*, 2010.
[39] E. Jonas, Q. Pu, S. Venkataraman, I. Stoica, and B. Recht. Occupy the Cloud: Distributed Computing for the 99%. In *Proc. ACM Symposium on Cloud Computing*, 2017.
[40] S. Kamburugamuve, S. Ekanayake, M. Pathirage, and G. Fox. Towards High Performance Processing of Streaming Data in Large Data Centers. In *Proc. IEEE International Parallel and Distributed Processing Symposium Workshops*, 2016.
[41] V. Kawadia and P. Kumar. A Cautionary Perspective on Cross-layer Design. *IEEE Wireless Communications*, 12(1), 2005.
[42] A. Khrabrov and E. de Lara. Accelerating Complex Data Transfer for Cluster Computing. In *Proc. 8th USENIX Workshop on HotCloud*, 2016.
[43] S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. M. Patel, K. Ramasamy, and S. Taneja. Twitter Heron: Stream Processing at Scale. In *Proc. ACM SIGMOD*, 2015.
[44] O. Michel, M. Coughlin, and E. Keller. Extending the Software-Defined Network Boundary. *ACM SIGCOMM Computer Communication Review*, 44(4):381–382, 2014.
[45] R. Moore. A Universal Dynamic Trace for Linux and other Operating Systems. In *Proc. USENIX ATC, FREENIX Track*, 2001.
[46] M. A. U. Nasir. Fault Tolerance for Stream Processing Engines. arXiv preprint arXiv:1605.00928, 2016.
[47] M. A. U. Nasir, G. D. F. Morales, D. García-Soriano, N. Kourtellis, and M. Serafini. The Power of Both Choices: Practical Load Balancing for Distributed Stream Processing Engines. In *Proc. IEEE International Conference on Data Engineering*, 2015.
[48] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari. S4: Distributed Stream Computing Platform. In *Proc. IEEE International Conference on Data Mining Workshops*, 2010.
[49] Z. Qian et al. TimeStream: Reliable Stream Computation in the Cloud. In *Proc. EuroSys*, 2013.
[50] R. K. Sahoo, A. Sivasubramaniam, M. S. Squillante, and Y. Zhang. Failure Data Analysis of a Large-Scale Heterogeneous Server Environment. In *Proc. IEEE International Conference on Dependable Systems and Networks*, 2004.
[51] M. A. Shah, J. M. Hellerstein, S. Chandrasekaran, and M. J. Franklin. Flux: An Adaptive Partitioning Operator for Continuous Query Systems. In *Proc. IEEE International Conference on Data Engineering*, 2003.
[52] R. Sherwood et al. FlowVisor: A Network Virtualization Layer. In *OpenFlow Switch Consortium*, 2009.
[53] M. Slee, A. Agarwal, and M. Kwiatkowski. Thrift: Scalable Cross-Language Services Implementation. *Facebook White Paper*, 5(8), 2007.
[54] M. Stonebraker, U. Çetintemel, and S. Zdonik. The 8 Requirements of Real-Time Stream Processing. *ACM SIGMOD Record*, 34(4), 2005.
[55] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, et al. Storm @Twitter. In *Proc. ACM SIGMOD*, 2014.
[56] L. Wang. Usages and Optimizations of Spark at Tencent. In *Proc. Big Data Conference*, 2015.
[57] B. White et al. An Integrated Experimental Environment for Distributed Systems and Networks. *ACM SIGOPS Operating Systems Review*, 36(SI), 2002.
[58] Y. Xing, S. Zdonik, and J.-H. Hwang. Dynamic load distribution in the borealis stream processor. In *Proc. IEEE International Conference on Data Engineering*, 2005.
[59] H. Zhang, G. Ananthanarayanan, P. Bodik, M. Philipose, P. Bahl, and M. J. Freedman. Live Video Analytics at Scale with Approximation and Delay-Tolerance. In *Proc. USENIX NSDI*, 2017.
[60] Q. Zhao, R. Rabbah, S. Amarasinghe, L. Rudolph, and W.-F. Wong. How to Do a Million Watchpoints: Efficient Debugging Using Dynamic Instrumentation. In *Proc. International Conference on Compiler Construction*, 2008.