IntStream: An Intent-driven Streaming Network Telemetry Framework

Xin Cheng*†, Zhiliang Wang*†, Shize Zhang*†, Xin He*†, and Jiahai Yang*†‡
*Institute for Network Sciences and Cyberspace, Tsinghua University, Beijing, China
†Beijing National Research Center for Information Science and Technology
‡Cyberspace Security Research Center, Peng Cheng Laboratory, Shenzhen 518000, China
Email: x-cheng19@mails.tsinghua.edu.cn, wzl@cernet.edu.cn, zsz16@mails.tsinghua.edu.cn
hexin21@mails.tsinghua.edu.cn, yang@cernet.edu.cn

***Abstract*—Due to the complexity of the network structure and the high growth of the transmission speed, the measurement and management of the network are facing serious challenges. The traditional bottom-up network telemetry methods are no longer applicable to complex network scenarios. To bridge this gap, we propose IntStream, an intent-driven streaming network telemetry framework to allow network operators to measure and analyze network traffic. However, there are three key challenges to building an intent-based telemetry system: (1) The diversity of network data sources. (2) The complexity of the measurement tasks. (3) The low overhead requirements of the telemetry system. IntStream introduces a lightweight component to extract and parse data from various types of data sources to form a data stream and divides the data stream conversation process into local and global stages. IntStream provides a set of rich expressive primitives to support users to write telemetry tasks based on intent. By performing part of the telemetry task on the local stage, the transmission overhead of intermediate data can be effectively reduced. The evaluation results conducted on a large campus network show that IntStream can support a wide range of telemetry tasks while reducing the intermediate data transmission overhead by** 99.64% **on average.**

***Index Terms*—Network Telemetry, Intent-driven.**

I. Introduction

Nowadays, it is increasingly challenging to make management decisions to improve the performance, availability, security, and efficiency for large-scale networks (e.g., enterprise, data center) [1]. To better understand the network to assist management decision-making, there have been growing interests in *network telemetry*. According to the definition of IETF [2], network telemetry acquires and utilizes network data remotely for network monitoring and operation. It helps operators gain better network insights and promotes efficient and automated network management.

Traditionally, network operators use a *bottom-up* approach to perform network telemetry. They use specific network monitoring tools (e.g., NetFlow [3]) to collect network data and infer the network-wide state based on these data. This approach will cause the scope of telemetry tasks to be limited to the underlying data collection tools, and usually is a highly manual process that requires operators' expertise. Hence, network telemetry should be performed in a *top-down* manner [1]. The network telemetry system should provide a query interface at the upper layer, which provides a set of primitives to support the operator to define telemetry tasks based on intent. However, building such an intent-driven network telemetry system faces several key challenges as follows:

- ***Challenge 1*: Flexibility.** The telemetry system needs to obtain data from various data sources (e.g., hosts, switches) in the network. Each type of data source has its own data extraction and parsing methods. How to be compatible with various types of data sources in a flexible way and shielding these details from operators is challenging. Existing telemetry systems usually obtain network data from specific data sources and lack a flexible mechanism to be compatible with other types of data sources [4]–[10].
- ***Challenge 2*: Expressiveness.** For operators to write telemetry tasks based on intent, the system must provide a set of primitives with rich expressiveness. Existing telemetry systems either only provide some domain-specific primitives to support limited telemetry tasks [8]–[10], or provide a set of primitives similar to stream processing (e.g., map, filter) to support some general stream processing operations [4]–[6]. But few telemetry systems can support custom processing logic.
- ***Challenge 3*: Scalability.** Since telemetry tasks usually need to be performed in a large network, existing telemetry systems generally require a global analysis server (or cluster) [4], [8], [9], [11], [12]. As the scale of telemetry tasks continues to grow, the transmission overhead of intermediate data and the load of the global analysis server will become the bottleneck of the entire telemetry system.

In this work, we propose IntStream, an intent-driven streaming network telemetry framework, which addresses all of the above challenges. IntStream divides the telemetry task processing into three stages: (1) a lightweight component, named driver (detailed in Section IV-B), is used in the data stream extraction stage to be compatible with various types of data sources in a flexible way. (2) In the local stream processing stage, part of the telemetry task is executed at various data sources in the network to generate local measurement results. (3) In the global stream processing stage, local measurement

978-3-903176-36-2 ©2021 IFIP

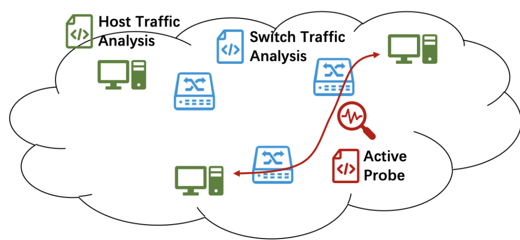

Fig. 1. Various types of data sources.

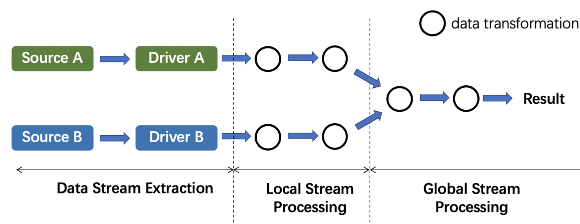

Fig. 2. Data stream processing stage.

results are gathered and performed global analysis to generate the final results. As part of the telemetry tasks are completed locally at the data source, the transmission overhead of intermediate data and the load of the global analysis server can be effectively reduced. In addition, IntStream provides a unified and layered programming model (detailed in Section III) in both the local and global processing stages, providing operators with a set of primitives with rich expressiveness to write a series of complex network telemetry tasks based on intent. The contributions of this paper are as follows:

- We propose a set of unified and layered network telemetry primitives. Operators can use these primitives with different expressiveness to define the processing logic in both the local and global processing stages based on their intentions.
- We design and implement a network telemetry framework called IntStream, which can be compatible with different types of data sources in a flexible way. And we used 10 queries to prove that IntStream has strong expressiveness and scalability. The evaluation results show that IntStream can reduce the data transmitted to the global analysis server by 99.64% on average.
- We implement the prototype of IntStream [13] with 5173 lines of code (LoC) and deploy it in our campus network for over 6 months.

II. Motivation

As presented in Fig. 1, when network operators perform network-wide telemetry tasks, they may need to obtain data from various types of data sources in the network. For example, network operators may need to analyze data from the hosts, such as online network interface card (NIC) traffic or PCAP files. They may also need to analyze switch-side traffic (e.g., mirrored switch traffic). Besides, operators may send some active probing packets (e.g., ping or traceroute) and collect these results for further analysis. Different data sources have different ways to extract and parse data. So IntStream should introduce a lightweight component to be compatible with these various data sources and be able to shield these processing details from the operator. At the same time, in order to reduce the transmission overhead, more analysis and processing should be done locally after the data is extracted from the data source. In addition, to perform network-wide telemetry, there should be a global analysis server (or cluster) that gathers the local measurement results from the network and performs global analysis.

Hence, the key idea of IntStream is to divide the data stream processing into 3 stages, as presented in Fig. 2:

Data stream extraction. At this stage, the data streams need to be extracted from different types of data sources. IntStream introduces a lightweight component called driver. As shown in Fig. 2, each type of data source has a specific matching driver to extract the data stream from it.

Local stream processing. The data stream enters the local stream processing stage after extraction. At this stage, the data stream will be processed by several transformations (processing logic defined by the operator) to form the local measurement result.

Global stream processing. Local measurement results at different locations in the network will be transmitted to the global analysis server (or cluster). Similar to the local stream processing stage, it is further processed by several transformations to generate the final measurement result.

The driver is used to extract data streams from different types of data sources, which is imperceptible to operators (addressing *Challenge 1*). After extraction, the data stream has a similar format. IntStream can provide a unified set of primitives in both the local and global stages to support operators to write telemetry tasks based on intent (addressing *Challenge 2*). And by performing the telemetry task locally as much as possible, the transmission overhead of intermediate results and the load of the global analysis server can be effectively reduced (addressing *Challenge 3*).

III. Unified and Layered Programming Model

In this section, we will introduce the programming model of IntStream, which provides the primitives that can support operator to define the processing logic of telemetry tasks.

A. Overview

Unified. According to the data stream processing model (Section II), the telemetry task will be executed in two stages: local stream processing and global stream processing. Since the global stream processor needs to aggregate the local measurement results from various locations in the network. In order to meet higher throughput and lower processing delay, IntStream utilizes the distributed streaming framework, Apache Flink [14], as the global stream processor. Although the underlying execution of these two stages is different, IntStream provides a unified programming model that allows

TABLE I
SUMMARY OF PRIMITIVES

Type	Primitive	Description	Expressive-ness
Core Primitives	*key()*	Set the aggregation granularity of the stream	Strong
	window()	Set the time window	
	transform()	Custom processing logic	
Streaming Primitives	*filter()*	Filter out the items that meet the given conditions	Medium
	map()	Convert one item into another item	
	flatMap()	Convert one item into 0 or more items	
High-level Primitives	*sketch()*	API related to the sketch data structure	Weak
	bloomFilter()	API related to the bloom filter structure	
	sendProbe()	API related to performing active probing	

operators to use the same primitives to define the processing logic in both stages.

Layered. As shown in Table I, IntStream provides a set of primitives with different expressiveness. When operators need to write some highly customized telemetry tasks, they can use the underlying core primitives, which can customize the processing logic in the *transform()* primitive. The core primitives have the highest expressiveness, but operators need to write relatively more code. On the contrary, if they just want to write some common telemetry tasks, they can use high-level primitives to reduce the amount of code. But correspondingly, the expressiveness of the high-level primitives is relatively low. In short, operators can flexibly combine different levels of primitives to write the telemetry tasks according to their intentions.

B. IntStream Primitives

Specifically, as shown in Table I, the primitives of IntStream have three levels:

Core Primitives. The underlying core primitives provide a set of primitives commonly used in network telemetry tasks. *window()* specifies the length of time window (e.g., 10 seconds) to divide the data stream. *key()* specifies the aggregate granularity of the traffic. Network traffic is generally aggregated at any granularity of 5-tuples (e.g., source IP and destination IP). But IntStream can aggregate the traffic based on any fields in the data stream, such as packet length, TCP flags, etc. After the data is aggregated, operators can freely define the processing logic in the *transform()* primitive.

Streaming Primitives. Since network telemetry tasks are similar to stream processing tasks, some network telemetry systems (e.g., Sonata [4] and MAFIA [5]) provide primitives similar to those commonly used in stream processing frameworks (e.g., Flink [14] and Spark Streaming [15]). IntStream also provides streaming primitives. For example, *map()* can convert each item in the data stream into another item, and the logic of how to perform the conversion is defined by the operator.

```
1  stream.filter(pkt->pkt.get(TCP.flags).equals(S))
2      .key(pkt -> pkt.get(IP.dst))
3      .window(10)
4      .transform((elements, out) -> {
5          cnt = 0
6          for(pkt: elements)
7              cnt++
8          if(cnt > threshold)
9              out.collect(pkt.get(IP.dst))
10     })
```

. Query 1: Detect Newly Opened TCP Connections.

High-level Primitives. IntStream also provides high-level primitives to provide data structures and operations that are commonly used in network telemetry tasks. Laffranchini et al. [5] summarized the common network measurement tasks and found that sketch [16] and bloom filter [17] are two commonly used data structures. Therefore, IntStream provides operations related to these two data structures, such as *update* and *query* operations in sketch. When operators are writing network telemetry tasks, they can directly use the relevant primitives of these data structures in the *transform()* primitives. In addition, IntStream also provides primitives to perform active probing. Operators can use *sendProbe()* in the local stream processing stage to make the driver send out the corresponding active probing packet (e.g., ping) and return the results.

Query 1 gives an example of a simple telemetry task which detects hosts that have too many recently opened TCP connections. Query 1 first uses *filter()* to filter out all SYN packets (line 1). It then uses *key()* to specify the aggregation granularity of the stream as the destination IP and specifies the time window of the stream as 10 seconds through *window()* (line 2-3). At this point, the data stream will be divided by a 10-second window and all items with the same destination IP will be aggregated to apply the *transform()* primitive. Note that the *transform()* primitive contains two parameters: *elements* includes all aggregated items in the window and *out* is the result data stream. The result can be collected through the *out.collect()* operation. Therefore, Query 1 counts the total number of all SYN packets (line 5-7). If it exceeds the set threshold, the destination IP will be output as the result (line 8-10).

IV. INTSTREAM DESIGN

A. Overview

Fig. 3 presents the architecture of IntStream. The telemetry tasks (or queries) written with IntStream primitives will be submitted to Runtime. It then sends the subtasks to the corresponding drivers and Global Stream Processor (GSP). After the driver receives the telemetry task, it will extract and parse the required data stream from the corresponding data source and send it to the Local Stream Processor (LSP). LSP performs local telemetry tasks in the local processing stage and generates local measurement results. GSP aggregates these local measurement results (through Apache Kafka [18]) and performs global telemetry tasks to generate the

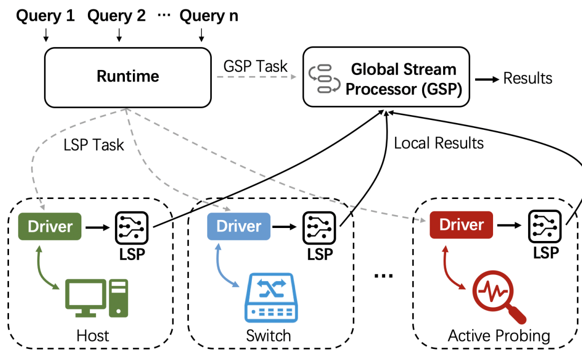

Fig. 3. IntStream architecture.

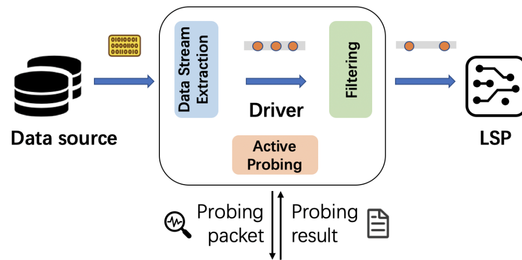

Fig. 4. Driver.

final results. Specifically, IntStream has the following main components:

Runtime. As the control component of the entire IntStream, Runtime is mainly responsible for task scheduling and the collection and display of the running status of the system. Operators can submit tasks through Runtime's web UI and view the execution of tasks. Runtime is also responsible for parsing the telemetry tasks and sending the subtasks to the corresponding drivers and GSP. Note that if the telemetry task can be completed in the local processing stage, there is no need for GSP to participate.

Driver (Section IV-B). The core function of the driver is to extract, parse and generate data streams from the data source. As mentioned in Section II, by developing different types of drivers, IntStream can be compatible with different types of data sources.

Local Stream Processor (Section IV-C). The LSP receives the data stream extracted from the driver and executes the processing logic defined by the operator in the local processing stage.

Global Stream Processor. By aggregating the local measurement results generated by the LSP, the GSP further executes the processing logic defined by the operator in the global processing stage. Similar to the existing telemetry system [4], [12], in order to meet higher throughput and lower processing delay, IntStream's GSP is implemented based on the existing big data framework Flink [14]. We will not introduce the detailed design of GSP in this section.

B. Driver

In general, there are various types of measurement data sources within the network (e.g., online NIC traffic and mirrored switch traffic). In order to be compatible with these different types of data sources in a flexible way, IntStream develops a lightweight component, named as driver, which is responsible for extracting, parsing, and filtering the original data from it, forming a data stream and sending it to the LSP for further processing. In addition, if active probing is required in the task, the driver will send the active probing packets and return the result to the LSP.

Specifically, in addition to some general operations (e.g., interacting with Runtime), the driver has the following three "pluggable" functional modules (presented in Fig. 4). Operators can implement their own modules as needed. For example, when operators need IntStream to be compatible with a new type of data source, they only need to implement a new data stream extraction module.

Data stream extraction. The driver extracts data from the original measurement data source and parses the required data fields. Taking the host's online NIC traffic as an example, the driver can use some packet capture libraries (e.g., libpcap [19]) to obtain the traffic data. IntStream implements a general data packet parsing module, which can parse common network protocols (e.g., the Ethernet layer, IP layer and TCP / UDP layer) from the original data packet content. Therefore, any driver can obtain the required fields (e.g., *IP.src* and *TCP.flags*) from the original packet content through this parsing module. In short, what is finally transmitted to the LSP can be regarded as a data stream where each data item contains several key-value pairs, e.g., $\langle timestamp = t, IP.src = x, IP.dst = y\rangle$.

Filtering. Generally, LSP does not need all the original data streams extracted by the driver. Operators usually use the *filter()* primitive first when writing telemetry tasks. For example, for Query 1 (in section III-B), the operator first filters out all SYN packets. If all data items are transmitted to the LSP and then performed the filter operation, it will cause excessive transmission and processing overhead. Hence, when the driver receives the telemetry task from Runtime, it will automatically perform the filtering operation defined by the operator in the first step (if any) after data extraction and parsing. So operators do not need to modify the driver code specifically to implement the filtering operation, but only need to define the filtering logic when writing the telemetry task.

Active probing. When operators need to perform an active probing task (e.g., obtaining the RTT of two hosts in the network), they can use the *sendProbe()* in the high-level primitives during the local processing stage to make the driver send out the customized probing packets. Specifically, taking the RTT of two hosts as an example, the operator can send 10 ping packets to a destination host and get the average RTT by calling $RTT = sendProbe(type = PING, dst = 1.2.3.4, num = 10)$. In addition to the active probing methods supported by IntStream, operators can also implement their own active probing primitives and use them

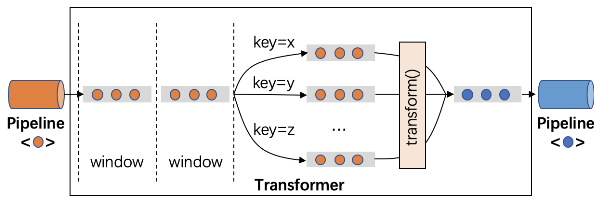

Fig. 5. Local Stream Processor.

under the IntStream framework.

C. Local Stream Processor

The LSP is usually on the same server as the measurement data source. If it needs to obtain the switch traffic, it will be deployed on a server close to the switch. It obtains the parsed and filtered data stream through the driver and performs customized data processing logic in the local processing stage. LSP is the core component of IntStream. By performing the telemetry task locally as much as possible, the transmission overhead of intermediate data and the workload of GSP can be effectively reduced.

As presented in Fig. 2, the data undergoes several transformations in the local processing stage, and each transformation converts a certain data type into another. Note that IntStream does not limit the scope of data types. It can be transformed into any user-defined data type during data processing to achieve rich expressiveness. Specifically, as presented in Fig. 5, LSP is composed of two main components: pipeline (responsible for data transmission) and transformer (responsible for performing data transformation):

Pipeline. The pipeline is a blocking first-in-first-out queue, which is responsible for transmitting data items between transformers. Each pipeline is assigned a data type, and all data items in the data stream transmitted in it are of this data type. As presented in Fig. 5, the data streams transmitted by the two pipelines at both ends of the transformer are of different types. Note that when the queue is empty, the transformer that reads data to it will fall into waiting. On the contrary, when the data in the pipeline exceeds the queue size, the transformer that outputs data to it will fall into waiting.

Transformer. To perform stream computing, each item in the data stream should enter the next transformer for calculation immediately after it is calculated in a certain transformer. Therefore, each transformer runs as an independent thread, which continuously obtains data from the input pipeline, performs corresponding data transformation and then outputs the result to the output pipeline. Each transformer always converts one type of data stream into another type of data stream. Fig. 5 presents the data processing of the transformer under the execution of the core primitives. First, it will divide the data stream according to the time specified by *window()*. All data items in the same window will be divided according to the key definition method specified by *key()*, and all data items with the same key value will be aggregated together. For example, $key(IP.src + IP.dst)$ combines the source IP and the destination IP as the key value. Then in each window, all data items with the same source IP and destination IP will be aggregated. For all these aggregated items, *transform()* will be performed to transform the data and output the result to the next pipeline.

```
1  drivers = [driver-1, driver-2, driver-3]
2  //local stream processing stage
3  for(driver: drivers){
4      stream=env.get(driver).getLocalStream()
5          .filter(pkt -> checkIpRange(pkt.get(IP.dst
               )))
6          .key(pkt -> pkt.get(IP.src))
7          .window(10)
8          .transform((elements, out) -> {
9              cnt = 0
10             for(pkt: elements)
11                 cnt++
12             out.collect(window.time, pkt.get(IP.src
                   ), cnt)
13         })
14         .sendToKafka(topic=sketch)
15 }
16 //global stream processing stage
17 stream2=env.getGlobalStream(topic=sketch)
18     .window(10)
19     .transform((elements, out) -> {
20         s = countMinSketch()
21         for(e: elements)
22             s.update(e[1], e[2])
23         out.collect(s)
24     })
```

. Query 2: Periodic sketch generation.

V. Case Study

In this section, we will show how operators write telemetry tasks under IntStream based on a complete example (Query 2). In this example, operators need to obtain the sketch data structure by the traffic generated from several measurement collection points in the campus network. Operators can further perform some anomaly detection tasks (e.g., heavy hitters) based on these sketches.

First, the operator needs to specify which driver to obtain data from (line 1). Note that the driver needs to be registered when joining IntStream so that the Runtime can interact with it. Since the operations performed by each LSP in the local processing stage are the same, the processing logic can be defined in a loop. Specifically, Query 2 uses *filter()* to limit the destination IP to the required network segment range (line 5). It sets the source IP as the key (line 6) and specifies the time window as 10 seconds (line 7). Query 2 performs the summation in *transform()* and sends the result to a certain topic in Kafka [18] (line 8 to 15). In the global processing stage, the GSP first obtains the local measurement results from Kafka (line 17) and uses *window()* to specify the time window as 10 seconds (line 18) and generates the sketch data structure in each window in *transform()*. Specifically, the operator first initializes a count-min sketch using *countMinSketch()* (high-level primitives) and then uses *update()* to update the sketch for each data item in the window (line 19-24).

TABLE II
THE FLEXIBILITY OF DRIVER.

Function Module	Custom Function	Description	LoC
Data stream Extraction	PCAP extraction	Extract data from PCAP file	113
	online traffic extraction	Extract data from NIC with packet capture library	154
	mirrored switch traffic extraction	Extract data from mirrored switch traffic captured by Stenographer [20]	249
Filtering	5-tuple filter	Support setting filter conditions on 5-tuples	135
	network segment filter	Support setting filter conditions on the range of IP address	13
Active Probing	ping	Perform ping probing to obtain RTT	95
	traceroute	Perform traceroute probing to obtain the routing information	166

VI. EVALUATION

In this section, we will evaluate IntStream from three aspects: flexibility, expressiveness and scalability.

A. Flexibility

The driver is designed as a lightweight component. In addition to some common operations (e.g., interacting with Runtime), it has three main functional modules that can be customized: data stream extraction, custom filter, and active probing. When operators need to perform some customized operations, they can selectively implement these functions.

Table II lists some of the driver functions that IntStream has currently implemented, which proves the strong flexibility of IntStream. In terms of data stream extraction, it can be seen that IntStream is compatible with common data sources in campus networks, such as online host traffic and mirrored switch traffic. Note that IntStream can also be compatible with non-traffic data (e.g., log files). In that case, operators need to implement their own data extraction and parsing logic to form a data stream containing the key-value pairs. In terms of custom filters, IntStream provides a universal 5-tuple filter, which can support common filtering operations on 5-tuples of data packets. If operators perform the filtering operation first in the telemetry task, IntStream will put it in the driver for execution to reduce the data transmitted to the LSP. In this way, the operator does not need to modify the code of driver specifically. In terms of active probing, IntStream currently provides two common active probing methods (ping and traceroute), which operators can call directly during the local stream processing stage. Operators can also implement their own active probing methods and execute them in the task.

B. Expressiveness

As presented in Table III, we used 10 telemetry tasks to demonstrate the expressiveness of IntStream. It can be seen that by combining several primitives, operators can define the processing logic of the entire telemetry task, including data extraction, parsing, filtering, processing, transmission, and result storage. Due to space constraints, please refer to [13] for the details of these 10 telemetry tasks. In addition, we also selected three typical telemetry systems and compared them with IntStream. MAFIA [5] performs the telemetry tasks in the programmable data plane, so the supported operations are limited. Sonata [4] makes comprehensive use of CPU and ASIC, but its primitives cannot support complex processing logic. dShark [11] uses CPU to process and analyze traffic like us and supports custom processing logic, but it only supports passive telemetry tasks.

However, the primitives provided by IntStream is very expressive, which can support complex custom processing logic. As can be seen from Table III, all telemetry tasks use core primitives (*key()*, *window()* and *transform()*). This is because the core primitives of IntStream are very consistent with the basic data processing logic of network telemetry. That is, the operator will first define the aggregation granularity of the data in time and space, and then define the processing logic on the aggregated data. At the same time, operators can also use streaming primitives to perform common stream calculations, or use common measurement data structures (e.g., sketch and bloom filter) and operations (e.g., active probing). In short, operators can write a series of complex telemetry tasks based on their intent.

C. Scalability

IntStream performs telemetry tasks by LSP and GSP. The underlying execution engine of GSP is Flink, a highly scalable stream processing engine. In addition, the design philosophy of IntStream is to perform the telemetry task locally as much as possible to reduce the transmission overhead of intermediate results and the load of GSP. Hence, we focus on evaluating the scalability of LSP.

Setup. We use the real traffic from our campus network (which has about 65 million hosts and 3.5 million switches) as the measurement data source and use Query 1 (Q1) and Query 2 (Q2) as the telemetry tasks. The server that LSP runs is equipped with Intel(R) Xeon(R) E5-2650 v2 CPU (2.60GHz) and 64GB memory. Specifically, we will evaluate the scalability of LSP from the following three aspects:

1) ***Data reduction****:* The processing flow of Q1 and Q2 are similar. First, a *filter()* is used to filter the original data stream and then *transform()* is used for custom processing. Therefore, we separately count the amount of data processed in these three stages (the amount of original data, the amount of data after *filter()*, and the amount of data after *transform()*) in each epoch (10 seconds). Fig. 6 and Fig. 7 present the data reduction of Q1 and Q2. It can be seen that the amount of data can be effectively reduced after filtering. Since IntStream optimizes the filtering operation to the driver, it can effectively reduce the amount of data transmitted to the LSP. At the same time, the amount of data after *transform()* will be greatly reduced. For Q1, the LSP outputs an average of 6223 items per epoch, which is 0.9% of the original data. For Q2,

TABLE III
SUMMARY OF APPLICATIONS.

#	Application	Description	Primitives used in IntStream	MAFIA [5]	Sonata [4]	dShark [11]	Int-Stream
1	TCP new conn (Query 1) [21]	Hosts for which the number of newly opened TCP connections exceeds threshold.	*filter(), key(), window(), transform()*	✓	✓	✓	✓
2	Sketch generation (Query 2) [16]	Generate periodic sketches of the traffic within a certain network segment.	*filter(), key(), window(), transform(), sketch()*	✓	✗	✓	✓
3	Port scan [22]	Hosts that send traffic over more than threshold destination ports.	*filter(), key(), window(), transform()*	✓	✓	✓	✓
4	Super spreader [23]	Hosts that contact more than threshold unique destinations.	*key(), window(), transform()*	✓	✓	✓	✓
5	TCP retransmission [24]	Find out frequently retransmitted TCP connections and get the RTT of them.	*filter(), key(), window(), transform(), flatMap(), sendPing()*	✗	✗	✗	✓
6	SYN flood [21]	Hosts for which the number of half-open TCP connections exceeds threshold.	*filter(), key(), window(), transform()*	✓	✓	✓	✓
7	RTT measurement	Obtain the periodic RTT between two hosts.	*filter(), window(), flatMap(), sendPing()*	✗	✗	✗	✓
8	Two-phase heavy hitter [5]	Two-phase heavy hitter detection based on sketch and bloom filter.	*window(), transform(), sketch(), bloomFilter()*	✓	✗	✓	✓
9	TCP incomplete flows [21]	Hosts for which the number of incomplete TCP connections exceeds threshold.	*filter(), key(), window(), transform()*	✓	✓	✓	✓
10	DDoS Detection [25]	DDoS detection based on sketch and hellinger distance	*key(), window(), transform(), sketch()*	✗	✗	✓	✓

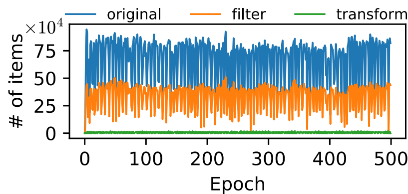

Fig. 6. Data reduction of Q1.

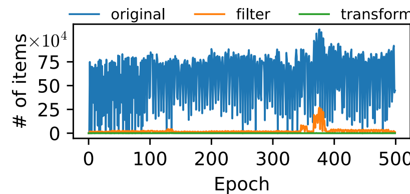

Fig. 7. Data reduction of Q2.

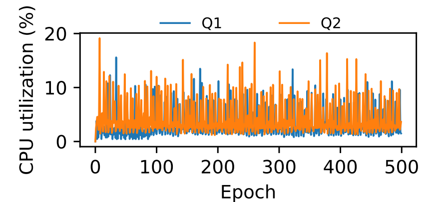

Fig. 8. CPU utilization of Q1 and Q2.

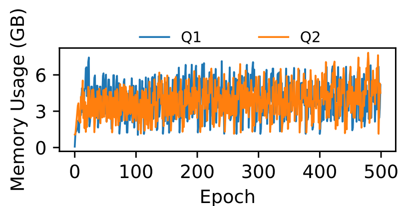

Fig. 9. Memory usage of Q1 and Q2.

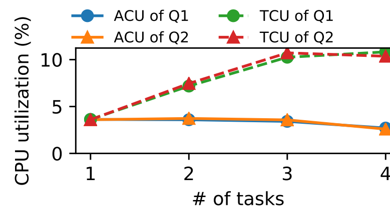

Fig. 10. CPU utilization under multiple tasks.

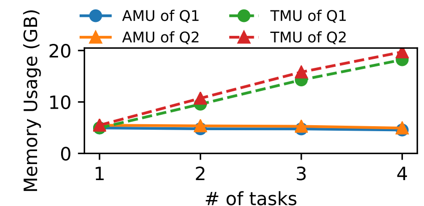

Fig. 11. Memory usage under multiple tasks.

the LSP outputs an average of 106 items per epoch, which is 0.018% of the original data. Although the specific ratio of data reduction is related to telemetry tasks, for all telemetry tasks in Table III, IntStream can reduce the data sent to GSP by 99.64% on average. Hence, LSP can greatly reduce the transmission consumption of intermediate data and the load of GSP, which improves the scalability of IntStream.

*2) **Resource consumption (single task):*** Since the LSP is usually deployed on the same host as the measurement data source, when the LSP performs the telemetry task, it should take up fewer resources (e.g., CPU and Memory) to avoid a significant impact on the collection of the measurement data or other programs. Therefore, we recorded the changes of the CPU utilization and memory usage during the execution of Q1 and Q2, as shown in Fig. 8 and Fig. 9. It can be seen that the CPU utilization rate of LSP is low when performing telemetry tasks, which is within 3% to 5% on average. In a few cases, the CPU utilization rate may exceed 10%. In addition, the memory consumed by the LSP is generally about 5GB, which is used for internal data transmission and processing. In general, LSP occupies fewer system resources when performing telemetry tasks and will not cause a significant impact on other tasks on the host.

*3) **Resource consumption (multiple tasks):*** If the administrator performs multiple telemetry tasks simultaneously, there may be several tasks running on the same LSP. Therefore, we evaluated the resource consumption by the LSP when it performs multiple tasks at the same time. Fig. 10

TABLE IV
COMPARISON WITH OTHER METHODS.

Telemetry System	Flexibility		Expressiveness	Scalability	
	Compatible to various data source	Support active probing	Expressiveness of the primitives	Support network-wide telemetry	Support local data processing
Trumpet [9]	✓	✗	Domain-specific	✓	✓
NetQRE [21]	✓	✗	Streaming	✓	✓
Sonata [4]	✗	✗	Streaming	✗	✓
Newton [7]	✗	✗	Streaming	✓	✓
Everflow [8]	✓	✓	Domain-specific	✓	✗
dShark [11]	✗	✗	Generic	✓	✗
IntStream	✓	✓	Generic	✓	✓

presents the CPU utilization of LSP under multiple tasks. It can be seen that as the number of tasks increases (less than 4), the average CPU utilization (ACU) of each task basically stabilizes at 3.5%. When the number of tasks is 4, it decreases to 2.5%. The total average CPU utilization (TCU) of the LSP increases as the number of tasks increases. Fig. 11 presents the memory usage of LSP under multiple tasks. It can be seen that as the number of tasks increases, the average memory usage (AMU) of each task is basically stable at about 5GB, and the total average memory usage (TMU) of the LSP increases as the number of tasks increases. In general, when the number of parallel tasks is less than 4, its performance is relatively stable. Hence, in practice, we can also control the number of parallel tasks of LSP within 4.

VII. Discussion and Limitation

At present, many telemetry systems are based on programmable switches. Thanks to ASICs, they can obtain the ability to process the data packets at a high speed. However, due to the limited functions provided by the programmable switch, it cannot provide operators with rich expressive primitives. To address the challenges mentioned in Section I, IntStream puts the LSP on the CPU to achieve strong flexibility and expressiveness.

Although IntStream cannot process the data packets at a high speed like ASICs, IntStream still makes many efforts to improve the scalability (e.g., data reduction at LSP). Even if each packet needs to be processed, usually only certain fields in the packet are needed for calculation. So in most cases, IntStream can achieve performance improvement through data reduction in the local stage. If IntStream needs to be compatible with some data sources with high throughput (e.g., switch), operators can use load balancing and deploy multiple drivers and LSPs. For example, we used 16 drivers and LSPs to process our campus traffic (up to 40 Gbps).

In short, compared to those telemetry systems based on programmable switches, IntStream has extremely strong flexibility and expressiveness, ensuring that administrators can write a series of complex telemetry tasks based on their intentions. Although IntStream cannot rely on a single point to process the data packets at a very high speed, it can also achieve this goal by adding hardware.

VIII. Related work

Network telemetry. According to the main execution location of the telemetry task, the existing telemetry systems can be divided into: (1) End-host. These works [9], [10], [21], [26] use the resources and programmability of the end-host to perform network telemetry tasks. They use the CPU to achieve strong expressiveness but the packet processing rate is limited. (2) Programmable data plane. These methods either implement approximate measurement by using limited resources in the programmable switch [23], [27]–[32], or implement stream processing primitives (e.g., map, filter, window) in the programmable switch to support operators to define a series of complex telemetry tasks [4]–[7], [33], [34]. Due to the limited operations that programmable switches can support, the expressiveness of these telemetry systems is relatively weak. (3) Global analysis server (cluster). Some systems [8], [11], [12], [35]–[38] perform network-wide telemetry tasks in the global analysis server (cluster) of a large-scale network. Everflow [8] and dShark [11] trace the path of the network traffic by collecting mirrored packets. Vtrace [12] and NetSight [35] use programmable switches to add metadata to the packets to obtain the telemetry information. Unlike these works, IntStream is compatible with different measurement data sources and provides very powerful expressive primitives in both the local processing stage and the global processing stage, which can support operators to write a series of complex network-wide telemetry tasks.

Intent-driven primitives. In order to be able to support operators to write telemetry tasks based on intent, existing telemetry systems generally provide a set of primitives. But the expressiveness of each set of primitives is different: (1) Domain-specific primitives (DSP). These works [8]–[10] usually perform some specific telemetry tasks, so the primitives they provide are also highly customized. For example, Trumpet [9] uses predicates and some aggregate functions to detect whether certain events have occurred. (2) Stream processing primitives. These works [4]–[6] usually provides some primitives commonly used in stream processing (e.g., filter, map). Compared with DSP, it can perform some more general operations. But it cannot support operators to write more complex and customized processing logic. (3) Generic primitives. It has the highest expressiveness and can support

operators to freely define the processing logic. dShark [11] uses C++ callback function to process the aggregated traffic and supports user-defined processing logic. Different from these primitives, IntStream provides a unified and layered programming model. Operators can use core primitives to freely define the processing logic, or use streaming primitives or some high-level primitives (encapsulated measurement data structures and operations) to simplify the telemetry tasks.

Comparison. As presented in Table IV, we have selected some representative works from the three types of telemetry systems and compared them with IntStream in terms of flexibility, expressiveness, and scalability. It can be seen that the existing works have varying degrees of deficiencies in these three aspects. Unlike these systems, IntStream is an intent-driven network telemetry framework that satisfies flexibility, expressiveness, and scalability simultaneously.

IX. Conclusion

In this paper, we propose IntStream, an intent-driven streaming network telemetry framework. IntStream uses a lightweight component, driver, to be compatible with various types of data sources. And IntStream provides a set of powerful expressive primitives which support operators to define the processing logic in both local and global processing stages based on their intentions. As part of the telemetry tasks are completed in the local processing stage, the transmission overhead and the load of the global analysis server can be effectively reduced. The evaluation results show that IntStream can support a wide range of telemetry tasks while reducing the intermediate data transmission overhead by 99.64% on average.

Acknowledgment

This work is supported by the National Key Research and Development Program of China under Grant 2018YFB1800200.

References

[1] M. Yu, "Network telemetry: Towards a top-down approach," *SIGCOMM Comput. Commun. Rev.*, 2019.

[2] H. Song. Network telemetry framework. https://tools.ietf.org/html/draft-ietf-opsawg-ntf-07.

[3] Cisco. Netflow. https://www.cisco.com/c/en/us/products/ios-nx-os-software/ios-netflow/index.html.

[4] A. Gupta, R. Harrison, M. Canini, N. Feamster, J. Rexford, and W. Willinger, "Sonata: Query-driven streaming network telemetry," in *Proc. of SIGCOMM*, 2018.

[5] P. Laffranchini, L. Rodrigues, M. Canini, and B. Krishnamurthy, "Measurements as first-class artifacts," in *Proc. of INFOCOM*, 2019.

[6] 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 *Proc. of SIGCOMM*, 2017.

[7] Y. Zhou, D. Zhang, K. Gao, C. Sun, J. Cao, Y. Wang, M. Xu, and J. Wu, "Newton: Intent-driven network traffic monitoring," in *Proc. of CoNEXT*, 2020.

[8] Y. Zhu, N. Kang, J. Cao, A. Greenberg, G. Lu, R. Mahajan, D. Maltz, L. Yuan, M. Zhang, B. Y. Zhao, and H. Zheng, "Packet-level telemetry in large datacenter networks," in *Proc. of SIGCOMM*, 2015.

[9] M. Moshref, M. Yu, R. Govindan, and A. Vahdat, "Trumpet: Timely and precise triggers in data centers," in *Proc. of SIGCOMM*, 2016.

[10] P. Tammana, R. Agarwal, and M. Lee, "Simplifying datacenter network debugging with pathdump," in *Proc. of OSDI*, 2016.

[11] D. Yu, Y. Zhu, B. Arzani, R. Fonseca, T. Zhang, K. Deng, and L. Yuan, "dshark: A general, easy to program and scalable framework for analyzing in-network packet traces," in *Proc. of NSDI*, 2019.

[12] C. Fang, H. Liu, M. Miao, J. Ye, L. Wang, W. Zhang, D. Kang, B. Lyv, P. Cheng, and J. Chen, "Vtrace: Automatic diagnostic system for persistent packet loss in cloud-scale overlay network," in *Proc. of SIGCOMM*, 2020.

[13] X. Cheng. Intstream. https://github.com/rickchengx/IntStream.

[14] T. A. S. Foundation. Apache flink. https://flink.apache.org/.

[15] ——. Spark streaming. https://spark.apache.org/streaming/.

[16] G. Cormode and S. Muthukrishnan, "An improved data stream summary: the count-min sketch and its applications," *Journal of Algorithms*, 2005.

[17] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, 1970.

[18] T. A. S. Foundation. Apache kafka. http://kafka.apache.org/.

[19] L. MartinGarcia. Libpcap. https://www.tcpdump.org/.

[20] google. Stenographer. https://github.com/google/stenographer.

[21] Y. Yuan, D. Lin, A. Mishra, S. Marwaha, R. Alur, and B. T. Loo, "Quantitative network monitoring with netqre," in *Proc. of SIGCOMM*, 2017.

[22] J. Jung, V. Paxson, A. Berger, and H. Balakrishnan, "Fast portscan detection using sequential hypothesis testing," in *Proc. of Symposium on Security and Privacy*, 2004.

[23] M. Yu, L. Jose, and R. Miao, "Software defined traffic measurement with opensketch," in *Proc. of NSDI*, 2013.

[24] E. Miravalls-Sierra, D. Muelas, J. Ramos, J. E. López de Vergara, D. Morató, and J. Aracil, "Online detection of pathological tcp flows with retransmissions in high-speed networks," *Computer Communications*, 2018.

[25] C. Wang, T. T. N. Miu, X. Luo, and J. Wang, "Skyshield: A sketch-based defense system against application layer ddos attacks," *TIFS*, 2018.

[26] K. Borders, J. Springer, and M. Burnside, "Chimera: A declarative language for streaming network traffic analysis," in *Proc. of USENIX Security*, 2012.

[27] Q. Huang, X. Jin, P. P. C. Lee, R. Li, L. Tang, Y.-C. Chen, and G. Zhang, "Sketchvisor: Robust network measurement for software packet processing," in *Proc. of SIGCOMM*, 2017.

[28] Q. Huang, P. P. C. Lee, and Y. Bao, "Sketchlearn: Relieving user burdens in approximate measurement with automated statistical inference," in *Proc. of SIGCOMM*, 2018.

[29] Y. Li, R. Miao, C. Kim, and M. Yu, "Flowradar: A better netflow for data centers," in *Proc. of NSDI*, 2016.

[30] X. Chen, S. Landau-Feibish, M. Braverman, and J. Rexford, "Beaucoup: Answering many network traffic queries, one memory update at a time," in *Proc. of SIGCOMM*, 2020.

[31] Z. Liu, A. Manousis, G. Vorsanger, V. Sekar, and V. Braverman, "One sketch to rule them all: Rethinking network flow monitoring with univmon," in *Proc. of SIGCOMM*, 2016.

[32] T. Yang, J. Jiang, P. Liu, Q. Huang, J. Gong, Y. Zhou, R. Miao, X. Li, and S. Uhlig, "Elastic sketch: Adaptive and fast network-wide measurements," in *Proc. of SIGCOMM*, 2018.

[33] R. Teixeira, R. Harrison, A. Gupta, and J. Rexford, "Packetscope: Monitoring the packet lifecycle inside a switch," in *Proc. of SOSR*, 2020.

[34] S. Wang, C. Sun, Z. Meng, M. Wang, J. Cao, M. Xu, J. Bi, Q. Huang, M. Moshref, T. Yang, H. Hu, and G. Zhang, "Martini: Bridging the gap between network measurement and control using switching asics," in *Proc. of ICNP*, 2020.

[35] N. Handigol, B. Heller, V. Jeyakumar, D. Mazières, and N. McKeown, "I know what your packet did last hop: Using packet histories to troubleshoot networks," in *Proc. of NSDI*, 2014.

[36] C. Guo, L. Yuan, D. Xiang, Y. Dang, R. Huang, D. Maltz, Z. Liu, V. Wang, B. Pang, H. Chen, Z.-W. Lin, and V. Kurien, "Pingmesh: A large-scale system for data center network latency measurement and analysis," in *Proc. of SIGCOMM*, 2015.

[37] A. Roy, D. Bansal, D. Brumley, H. K. Chandrappa, P. Sharma, R. Tewari, B. Arzani, and A. C. Snoeren, "Cloud datacenter sdn monitoring: Experiences and challenges," in *Proc. of IMC*, 2018.

[38] Y. Lin, Y. Zhou, Z. Liu, K. Liu, Y. Wang, M. Xu, J. Bi, Y. Liu, and J. Wu, "Netview: Towards on-demand network-wide telemetry in the data center," *Computer Networks*, 2020.