BENCHMARKING STREAM DATA PROCESSING SYSTEMS ON AN IOT INFRASTRUCTURE

Santosh Dhirwani
Technical University of Berlin
santoshdhirwani@yahoo.com

Yiming Gu
University of Technology, Sydney
Technical University of Berlin
kevinguemail@gmail.com

ABSTRACT

Internet of Things applications often require high performance, real-time data processing. Speed and data integrity are two vital attributes for most applications, for applications used in autonomous and connected vehicles as well as smart security for homes and businesses, these two attributes are particularly important to ensure that the applications can maintain their optimal performance. In addition, Internet of Things infrastructures are often distributed, moving and unstable, therefore, these applications need to have a Stream Data Processing System that can manage high-volume data on wireless networks with intermittent connectivity.

In this paper, we present the results of our microbenchmark on two state-of-the-art Stream Data Processing Systems - Apache Flink and Frontier - on an Internet of Things infrastructure, as well as an experimental analysis on the two systems. The outcome of the experimental analysis shows that Frontier's performance leads both in terms of latency as well as resource consumption, although Apache Flink is able to sustain higher throughput levels. A detailed evaluation has been included to compare and highlight the individual performance of each system.

1 INTRODUCTION

The Internet of Things, or IoT, is a network of interrelated devices that are able to exchange data without the need of any human-to-human or human-to-machine interactions. With recent technological advancements in IoT, its growth has garnered great momentum. The International Data Corporation projects the total IoT market size to grow to $1.1 trillion by 2023 [1]. This tremendous growth in market size is being propelled by the increasing number of IoT platforms entering the market, by the end of 2019 the total number of publicly known IoT platforms surpassed 620, more than double compared to 2015 [2]. Consequently, the number of IoT connected devices is also forecasted to increase exponentially to 41.6 billion by 2025, likewise, the projected data generation is also expected to exceed 79.4 zettabytes [3].

IoT provides the possibility for businesses to monitor their systems in real-time, it is increasingly utilised to gain better insights into the performance of their devices and operations. An IoT infrastructure consists of devices with embedded systems that are able to capture, transfer, comprehend and act on the data that they obtain from their surrounding environments. These devices can then share the data collected by their various sensors to the cloud or with other related devices, which can then act on the information provided.

The amount of data generated by IoT devices can be overwhelming, therefore, businesses need a way to manage and facilitate their data processing procedures. Conventional data processing systems save the incoming data in centralised storage centres, applications can then query or compute over the data when needed. However, IoT applications used in products such as autonomous driving cars or preventive maintenance require the ability to respond to sudden changes rapidly, thus, the data collected by these applications need to be analysed or processed in real-time. Stream Data Processing Systems (SDPS) aim to streamline the traditional data processing procedures. The data processing logic, queries and analytic functions work continuously as the incoming data streams flow through the systems round-the-clock. SDPSs have become an indispensable component in many businesses, as a vital mediator between device data and meaningful analyses or data operations, reliability and the ability to correctly process high-speed and high-volume data is crucial. In the case of IoT applications, current SDPSs are not fully capable of overcoming all the challenges that IoT presents [4]. Several IoT specific SDPSs have been introduced by researchers including NebulaStream [4] and Frontier [5], these IoT specific data processing systems aim to address the shortfalls of current SDPSs.

This research benchmarks the performance of Apache Flink and Frontier, two state-of-the-art SDPSs, in order to evaluate their suitability for use on an IoT infrastructure. An emulated IoT environment that is comparable to real-

world wireless networks has been set up to determine which SDPS can sustain higher throughput levels while maintaining reasonable resource consumption levels. The evaluation focuses on latency, throughput, CPU load and memory utilisation.

2 OVERVIEW

There are a number of established, high-performance SDPSs currently available, one such example is Apache Flink. This paper compares Apache Flink's performance with a relatively new framework - Frontier. In this section, we provide a brief overview of Flink and Frontier's architecture, as well as the benchmarking infrastructure that was used.

2.1 Apache Flink

Apache Flink is an open-source, scalable and distributed stream processing engine that is designed to run high-performance applications at any scale, anywhere [6]. It excels at processing both bounded and unbounded streams of data. Flink is capable of running large-scale, fault-tolerant applications while delivering real-time performance, an industry favourite, it is deployed in many companies around the world. It uses the parallelisation contract (PACT) programming model - a generalised version of MapReduce. Essentially, a Flink application consists of several tasks, each task is then split into multiple parallel instances each processing a subset of the main task's data. Flink's two main components are operators and streams, operators consume, transform and produce new streams from intermediate steams. Backpressure occurs when an operator receives data at a higher rate than it can process, if not handled correctly, backpressure can cause the system to exhaust its resources or even lead to data loss. Flink uses blocking queues with a bounded capacity to ensure that backpressure will be propagated throughout the entire pipeline, stream operators exchange intermediate results via buffers, an operator sends its buffer downstream when it is full or after a timeout [6]. This feature provides Flink with the exceptional ability to maintain its high-throughput and low-latency profile while processing an enormous volume of data.

Flink has been chosen as the first system under test due to its widespread adaptation as well as its benchmarking performance when compared to other SDPSs such as Apache Storm and Apache Spark [7].

2.2 Frontier

Frontier is an experimental, distributed and resilient edge processing framework designed for IoT devices [5]. It aims to provide a robust data processing system that can handle changes in the network bandwidth between different IoT

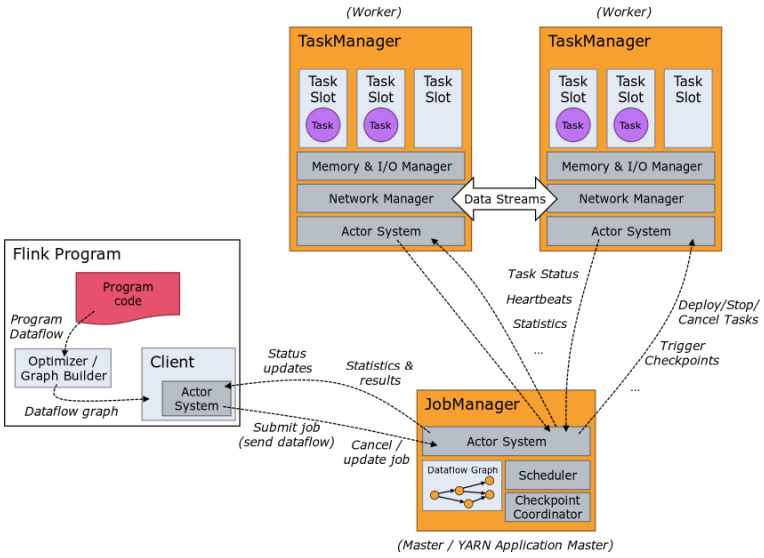


FIGURE 1: Apache Flink Architecture [6]

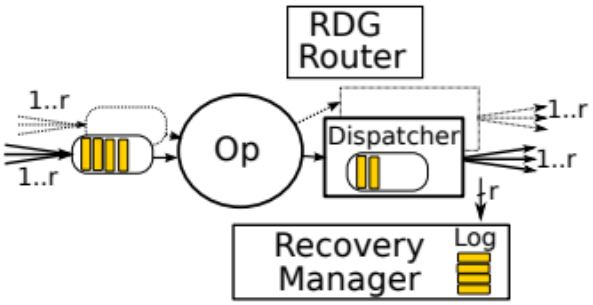


FIGURE 2: Frontier node architecture [5]

devices caused by either interference or connectivity issues. Due to the often unstable network connections and shifting bandwidth experienced by mobile, wireless networks, IoT devices in an edge deployment model often require the ability to perform data calculations or transformations locally without connecting to a cloud backend. Frontier transforms IoT applications into data-parallel, streaming queries, and exploits network path diversity in order to improve its throughput even with unstable wireless networks. It accomplishes this by replicating query operators at different nodes which provides more paths for network data. Frontier supports both Raspberry-Pi and emulation deployments. The three main features of Frontier are as follows:

1. Replicated dataflow graphs (RDG): N-ary replicated stateless or stateful processing operators across mul-

tiple nodes. Frontier dynamically routes data batches to downstream replicas which increases the probability of the network path being viable. RDG utilises all available paths when possible, maximising the network bandwidth.

2. Backpressure stream routing: Batches in Frontier are routed through the RDG using a novel network-aware stream backpressure routing (BSR) algorithm. Processing rate, congestion and quality at downstream replicas are scored using weights, which are then used by upstream replicas to route batches downstream.
3. Selective network-aware replay (SNAP): Processed batches are stored in each node's send buffer for retransmission and upstream nodes are notified of received batches selectively. In the event of a network disconnection, the unprocessed batches are resent by the upstream node to different operator replicas.

Frontier has been chosen as the second system under test due to claimed high performance in IoT environments [5].

2.3 CORE

Common Open Research Emulator (CORE) is a tool that can be used to emulate an IoT network. It is used to build and represent real computer networks that run in real-time, with the ability to connect to physical networks and routers. CORE provides an environment where real applications and protocols can be emulated and run, which makes prototyping and testing easier as well as providing valuable insights on network operations for research and demonstrations. CORE emulates layers 3 and above - network, session and application [8].

Figure 3 shows the architecture of CORE, at a glance, it consists of various interrelated components. The four main components are core-daemon, core-gui, coresendmsg and vcmd. For a given network, the core-daemon manages the emulated sessions, the core-gui, which allows users to create nodes and links via drag and drop can then be used to control and manipulate the session. Two command-line utilities - coresendmsg and vcmd, are used to send TLV (Type-length-value) API messages to the core-daemon and shell commands to nodes receptively [8].

2.4 EMANE

Extendable Mobile Ad-hoc Network Emulator (EMANE) is a network emulator that can be used in conjunction with CORE to emulate complex wireless radio models. While CORE is used to emulate layers 3 and above, EMANE is used to emulate layers 1 and 2 - physical and data link - using pluggable PHY and MAC models. It focuses on real-time modelling of these two layers so that applications can be experimentally subjected to conditions actual mobile, wireless networks would experience [9]. EMANE interfaces with CORE via a TAP device - a virtual network kernel interface supported entirely in software. First, CORE builds the virtual node using Linux network namespaces and installs the TAP device into the namespace, one EMANE process is then instantiated in the namespace. The EMANE process binds a userspace socket to the TAP device for sending and receiving data from CORE [10].

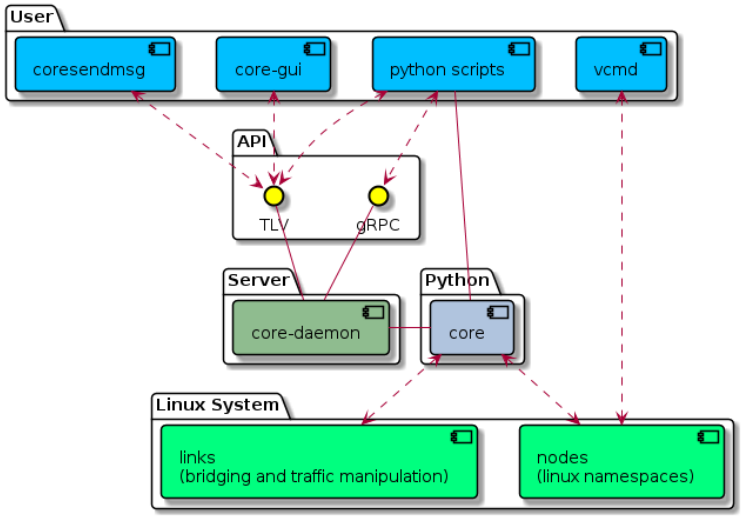


FIGURE 3: CORE architecture [8]

EMANE instances send and receive Over-The-Air (OTA) traffic to and from other EMANE instances via a control port (e.g. ctrl0, ctrl1). Configuration of the EMANE models is done through the WLAN configuration dialogue of CORE. A corresponding "EMANEModel" python class is sub-classed for each supported EMANE model to update XML files which stores configuration items and their mapping. In doing so, new models can be supported easily. CORE generates the relevant XML files that specify the EMANE NEM configuration on startup and launches the daemons. CORE and EMANE can also run in distributed environments among two or more emulation servers. It is similar to running them on a single machine and it can be achieved easily by adjusting a few key configuration parameters. Figure 4 shows the architecture of distributed EMANE.

3 OBJECTIVES

This research's contributions are threefold. Firstly, a micro-benchmark is defined in order to assess each System Under

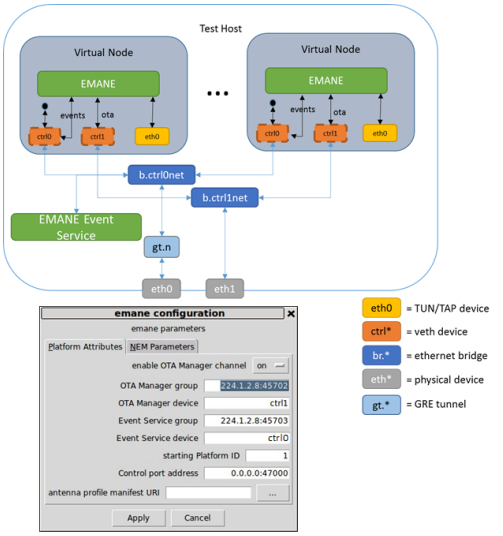


FIGURE 4: Distributed EMANE [10]

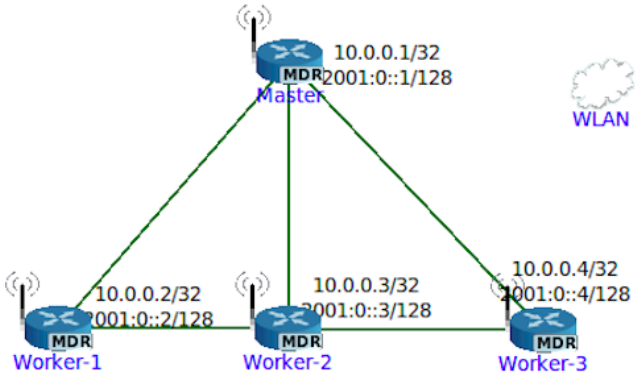


FIGURE 5: EMANE topology

Test (SUT). Secondly, an emulated IoT infrastructure is designed to assess the SUTs. Finally, an experimental analysis is carried out on each SUT for a more thorough comparison.

3.1 Microbenchmark

The first research objective is to define a micro-benchmark in order to evaluate the two SUTs on a small set-up, this is done to establish the baseline data processing performance for each system. State-of-the-art SDPSs such as Flink have been proven to scale to thousands of cores and terabytes of application state [6], a micro-benchmark can detect the nuances in their performance and is also easier to monitor and observe the results.

For the purpose of this research, each system is configured to run the same stateful query - a sliding window function - and listen on their ports for input data generated by the input generator. The microbenchmarking steps are as follow:

1. Varying the throughput in order to observe the changes in processing latency for each system.
2. Observing and evaluating the resource consumption for each system, in this case, memory utilisation and CPU load.

The results are shown on the console output to the user and also captured in an output file. We developed and used python scripts to plot the latency and system usage results.

3.2 Infrastructure

IoT infrastructures are often complex systems that can contain tens of thousands of devices all interrelated and interconnected to each other. This research uses CORE and EMANE in order to emulate a comparable IoT environment. As laid out in the previous section, the role of CORE is to emulate the virtual network that the SDPSs will run on. CORE is set up on a cluster along with the two SUTs.

CORE allows users to create custom network topologies either through its built-in Graphic User Interface (GUI) or through python scripting. Although the same topology could be set up through scripting, most of CORE's usage and source code base is based on the GUI. The scripting does not have as much documentation and has many quirks that are difficult to diagnose. Therefore, this paper will focus on using the GUI alone.

An EMANE network topology, shown in Figure 5, has been created to emulate an IoT infrastructure for the two SUTs. The topology consists of a WLAN node configured to run an EMANE process, and 4 routers that all connect to the EMANE network. The router on top is being used as the master node and the other three remaining nodes are being used as worker nodes. The number of worker nodes is easily scalable.

3.3 Experimental Analysis

The third contribution of this study is to perform an experimental analysis on Flink and Frontier. The goal of the analysis is to find out which of the two systems has the lowest latency, lowest resource consumption and highest sustainable throughput on an IoT infrastructure. The experiments were conducted in an isolated environment, where we benchmarked the two SUTs using the defined microbenchmark on the defined infrastructure.

4 IMPLEMENTATION

This section details how the various components required by the experiment were configured or setup, a detailed description of the workload design is also included to provide further insights into the experimental design.

4.1 CORE, EMANE and Services

As stated previously, in order to design and emulate a suitable IoT environment that is comparable to a real-world, wireless mobile network, CORE was used in conjunction with EMANE to build and emulate the testing ground for Flink and Frontier. CORE and EMANE were installed on the cluster provided by the Database Systems and Information Management (DIMA) group from the Technical University of Berlin - cloud-7. Once installed, the CORE daemon needed to be run with root privileges in order for the various experimental processes to function correctly.

Nodes emulated by CORE can be assigned custom functions, or services, as aptly named by CORE. Services are used by CORE to specify which scripts or processes run on nodes when they are started. The standard node types included by CORE have predefined services already assigned to them, users can then either customise existing services or define and assign new services to nodes to provide more functionalities as they see fit. To speed up the experimental process and make it not only reproducible but also easily configurable, a number of custom services were created and assigned to the MDR (MANET Designated Routers, runs Quagga OSPFv3 MDR routing for MANET-optimized routing) nodes shown in Figure 5. They help facilitate the execution and processing of experiments and ensure that operator errors are contained and limited.

4.2 Workload Design

The workload consists of the scalable input generator where the throughput was configured to generate a specific number of messages per second [11]. We changed the throughput from 100,000 to 1.5 million messages per second in steps of 250,00 messages per second. The max number of messages was set to 1.5 million messages per second in order to avoid excessive memory consumption. An Apache Flink job running an extension of the Yahoo Streaming Benchmark (YSB) was used [11]. Both SUTs process the input messages using a sliding window. The window size is 10 seconds whereas the window slide is 1 second. Also, the parallelism was changed varied as 2,3 and 4 for both systems.

4.3 Input Generator

The study uses a scalable input generator. It contains different types of input generators and the one used for this study is the netty benchmark generator [11]. Along with the throughput (number of messages per second) and the max number of messages, the sender port can be specified as well. In this study, the messages generated by this generator were forwarded to the two systems under test. The tcp/ip port is opened in both systems for listening to the messages sent by the input generator.

4.4 Systems Under Test

For the purpose of this research, we used a modified version of Apache Flink that consists of a job which runs the YSB and listens on the configured port for messages from a scalable input generator [11]. The Frontier job was developed to listen to the input generator in the same manner and to run the YSB. The EMANE topology was built and used for both SUTs having 1 master node and 3 worker nodes.

5 EXPERIMENTATION

In order to obtain results for this study, six sets of experiments were performed, three for each system, with seven executions in each set, to analyze the performance of the two SUTs. An isolated experimentation environment was successfully built to have the same setup for the two systems.

5.1 Data source and Cluster

We are using an Input Generator as the data source for both SUTs. The Input Generator generates a certain number of messages per second which can be configured. In order to change the throughput, one needs to change the number of messages per second. As an experimental environment, IBM Cluster is used which is provided to us by the DIMA group at Technical University of Berlin. The cluster has 4 physical machines but we are using only one for our study. The physical machine has 47 GB total memory and 16 cores. Operating system being used here is Ubuntu 18.04.2 LTS with Linux version 4.15.0-46-generic. As the Flink server, version 1.10 is being used.

5.2 Evaluation Metrics

The study aims to evaluate the performance of Apache Flink and Frontier on our defined IoT infrastructure. The evaluation is done based on two categories which are resource and network consumption. The evaluation metrics are latency, throughput, CPU load and memory utilization. Latency

is captured in milliseconds with reference to time which is captured in seconds. CPU load and memory utilization are the system usage metrics which are captured in terms of percentage. We define the type of latency and throughput that we are considering in this study.

Latency in state-of-the-art SDPSs can be divided in two categories, *event-time* latency and *processing-time* latency. Event-time is the time when an event is captured while processing-time is the time when an operator processes a tuple [7]. Event-time latency is defined as the time interval between a tuple's event time and its emission time from the SUT operator whereas processing-time latency is defined as the time interval between a tuple's ingestion time and its emission time from the SUT operator. The latency that we are focussing on, in this study, is event-time latency.

Throughput of a stream data processing system is the number of events that the system can process in a given amount of time.

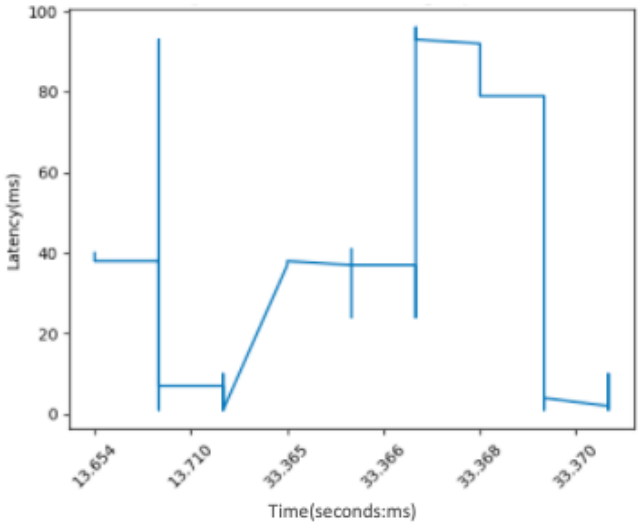
5.3 Experimental Configuration

The parameters that were tweaked are parallelism and throughput. Initially, we did some dry runs by varying the throughput in large steps. Initial experiments were performed for the throughput range of 50000 to 2 million messages per second. On observing the results of these initial experiments, the throughput range from 100k messages per second to 1.5 million messages per second yielded the most interesting sets of results which could give us insights into the performance of the two SUTs. The throughput was changed in steps of 250k messages per second whereas the parallelism was changed between 2,3 and 4. Frontier code snippets can be seen in Appendix A.

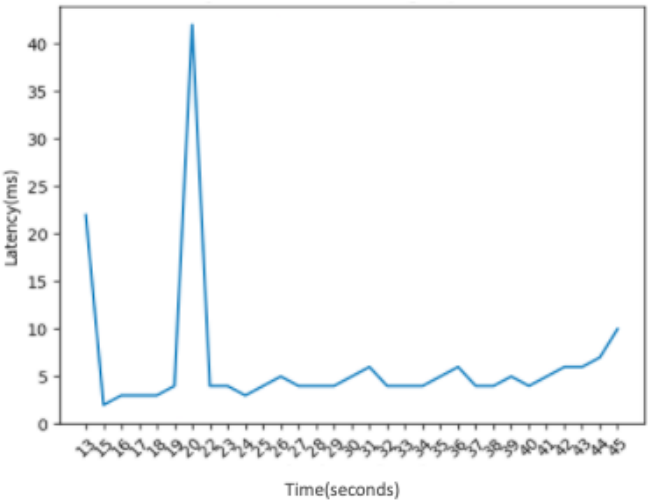
5.4 Experiments

In order to prove this study, experiments are performed to analyze the performance of the two SUTs by using the defined microbenchmark on the defined IoT infrastructure.

5.4.1 Determining the system with lowest latency. We were successful in achieving a set of very interesting results and in the following section, we are describing the experiments and their results in detail. Figure 6 shows the latency of the two SUTs at a throughput of 1 million messages per second. The x-axis shows the time in seconds and the y-axis shows the latency in milliseconds. As shown in the figure, the max latency of Apache Flink is 100ms whereas the max latency of Frontier is 40 ms. This peak in Frontier is due to the master node being started. The addition and running of 3 worker nodes does not have a huge impact on the latency of Frontier. We suspect that Apache Flink has higher



(a) Apache Flink

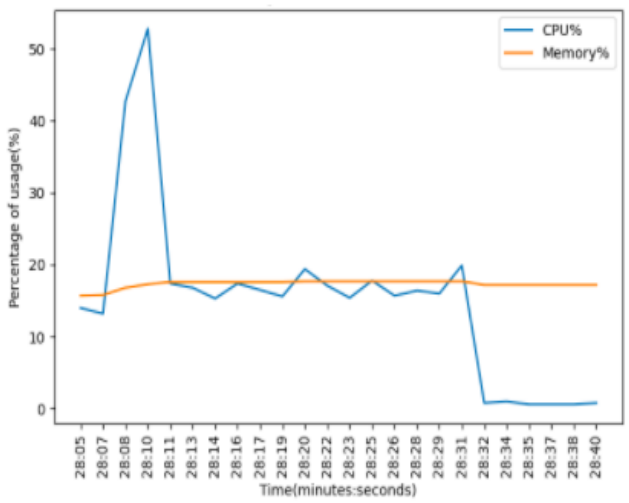


(b) Frontier

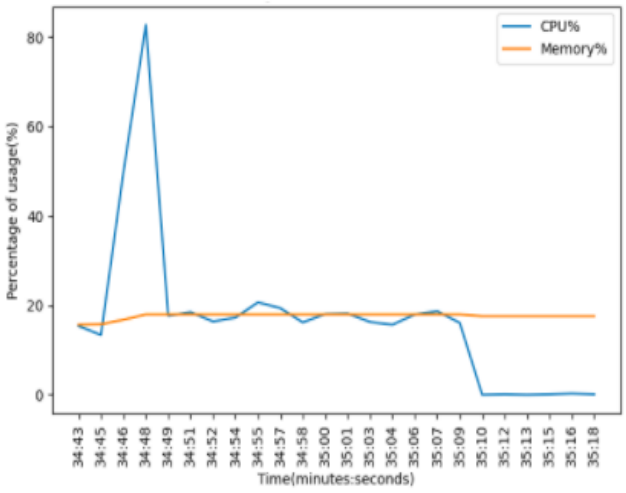
FIGURE 6: Latency plots of the two SUTs

latency than Frontier because it has a much more complicated and flexible engine, which is also very powerful. It does a lot of work that is not directly related to running the window query. On the other hand, Frontier is much simpler. It just receives data, processes it and sends it to the sink. However this needs to be investigated further.

5.4.2 Determining the system with lowest resource consumption. The two most important parameters in resource consumption are memory utilization and CPU load. When we observed the resource consumption for Apache Flink at throughputs of 750,000 and 1 million messages/second, as shown in Figure 7 the results were very interesting. The

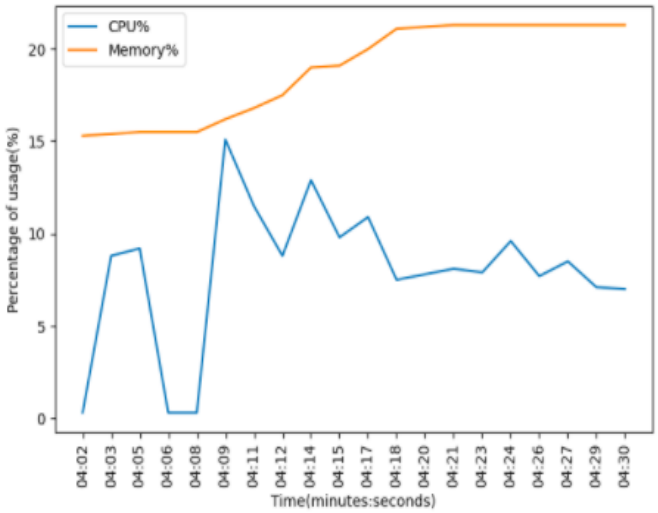


(a) Throughput: 750,000 messages/second

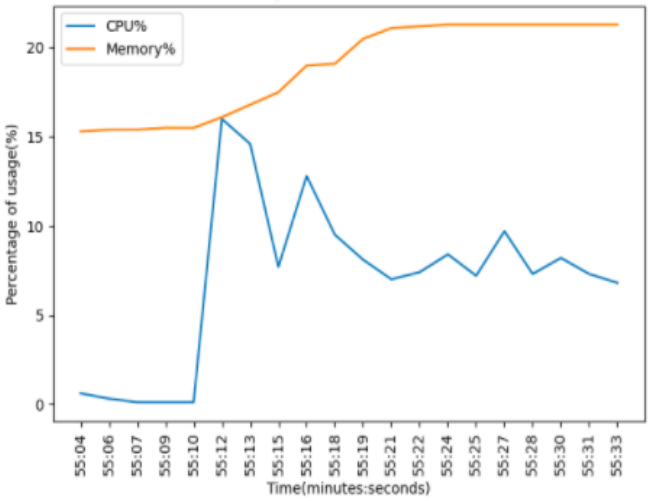


(b) Throughput: 1 million messages/second

FIGURE 7: Resource consumption of Apache Flink



(a) Throughput: 750,000 messages/second

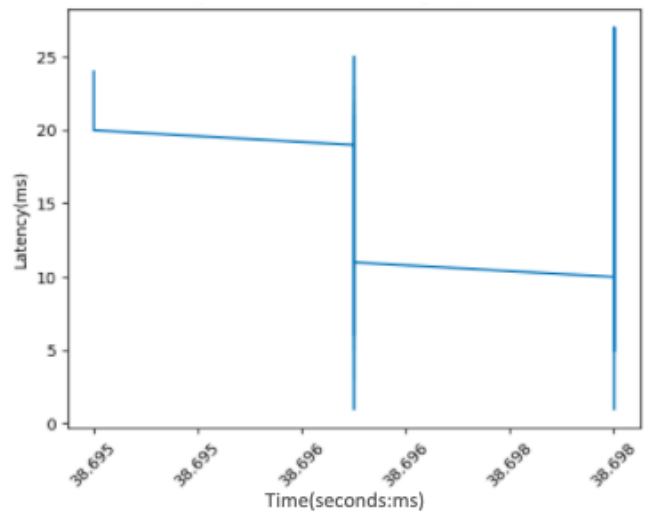


(b) Throughput: 1 million messages/second

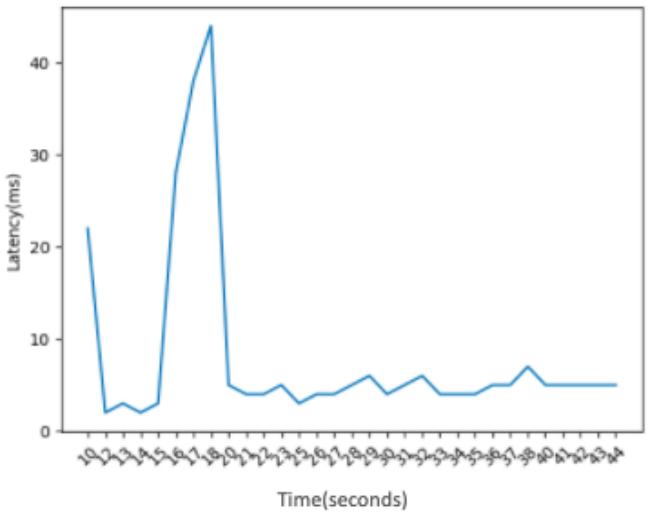
FIGURE 8: Resource consumption of Frontier

CPU load was 50% and 80% respectively which is not the best for an IoT infrastructure where the number of devices is usually very high and the throughput is much higher. However, for Frontier, we observed the resource consumption to be much lower for the same throughputs. The CPU load for Frontier was approximately 20% for throughputs of 750,000 and 1 million messages/second as shown in Figure 8.

5.4.3 Determining the system with highest sustainable throughput. To avoid large latency fluctuations is critical on an IoT infrastructure. The sustainable throughput is a parameter that can be considered to evaluate fluctuations of a system. Figure 9 shows the latency plots of both systems for a throughput of 750,000 messages/second. The latency of Apache Flink fluctuates between 0 to 25 milliseconds whereas the latency for Frontier fluctuates between 0 to 40 milliseconds. This high fluctuation of latency in Frontier can already be observed at a throughput of 250,000 messages per second as shown in Figure 10. Therefore, when higher sustainable throughput is a priority on an IoT infrastructure, Apache Flink is the better choice.



(a) Apache Flink



(b) Frontier

FIGURE 9: Latency fluctuations for throughput of 750,000 messages/second

5.4 Discussion

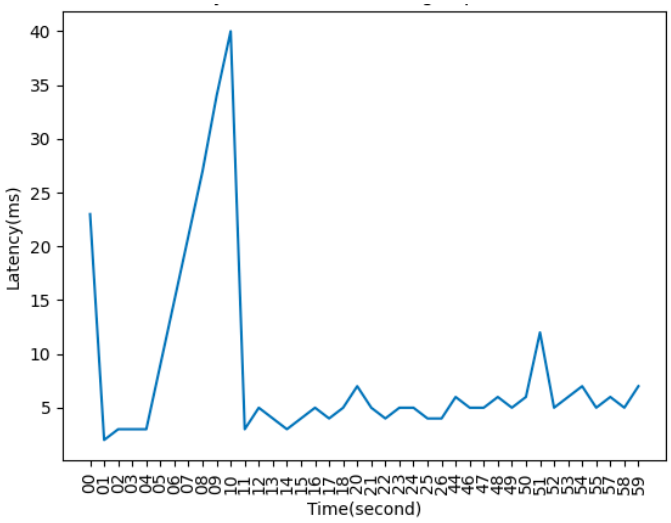
The key insights of evaluation of the two systems in our study are three-fold. First, we define a mircobenchmark for benchmarking SDPSs on an IoT infrastructure. The benchmark includes tweaking two parameters, namely parallelism and throughput. The output values for latency, timestamp, CPU load and memory utilisation are captured. We are using python scripts to plot graphs based on these output values. The experiments were performed for the parallelism values of 2,3 and 4 in the throughput range of 50,000 to 2 million messages per second. However, the most interesting results were obtained with the parallelism value of 3 and for the throughput range of 250,000 messages per second to 1 million messages per second, in the steps of 250,000 messages per second. Second, we define and emulate an IoT infrastructure using CORE and EMANE. The topology built and described in this study is used for both systems under test. It consists of one master node and three worker nodes. Third, we conduct a detailed experimental analysis of the two systems using the defined microbenchmark on the defined infrastructure. The results of the experimental analysis show that Frontier has lower latency and lower resource consumption when compared to Apache Flink whereas Apache Flink shows higher sustainable throughput than Frontier. We suspect that the reason for these results is the complexity and power of the Apache Flink engine which leads to higher resource consumption compared to Frontier. We also suspect that the BSR algorithm incorporated in Frontier is the reason behind it's better performance in terms of latency. However, the precise reasons behind these results still needs to be investigated by conducting a further intensive and detailed study on the two systems.

6 CONCLUSIONS AND FUTURE WORK

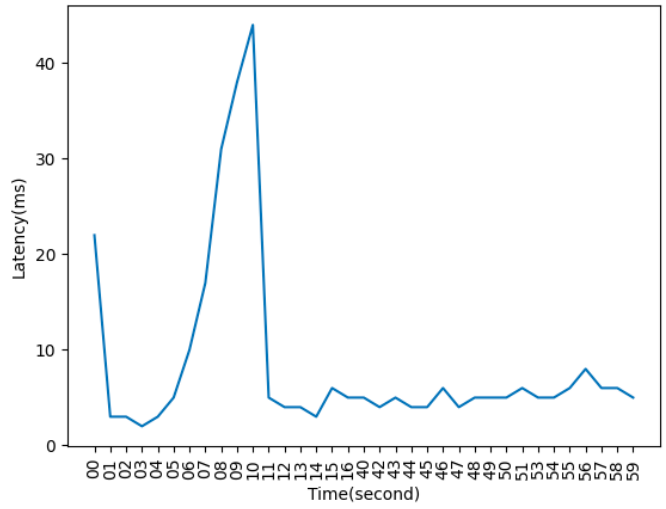
The study compares the performance of Apache Flink and Frontier on an IoT infrastructure in a detailed manner. Frontier has lower latency and lower resource consumption compared to Apache Flink while providing a higher throughput at the same time. Low resource consumption is a huge advantage on an IoT infrastructure as most of the devices are not very powerful. Apache Flink has higher sustainable throughput than Frontier.

This report has documented all the steps for implementation so that results can be reproduced and analyzed all experiment results in detail, which will be very useful for further studies based on evaluating data stream processing systems on IoT infrastructures. However, some questions still remain open for research. It can also add significant value to find out which system performs better when the throughput is varying rather than being constant. The varying throughput feature represents the capability of the SDPS to handle a huge amount of change in the number of devices on an IoT infrastructure. In order to cover more IoT use cases, experimentation by changing the number of nodes and tracking network consumption would be very valuable.

If it was possible to work longer on this project than the length of just one semester, we could have also evaluated the performance of the SUTs based on a tumbling window rather than just a sliding window. It would be very interesting to measure the performance of these systems with other



(a) Throughput: 250,000 messages/second



(b) Throughput: 500,000 messages/second

FIGURE 10: Latency fluctuations for Frontier

stateful queries queries as well rather than just windows, for example, join queries. We also thought about shaping this into a sophisticated and well automated command line tool. This can be achieved via bash and python scripts. As a part of this study, we developed two python scripts to plot results and two bash scripts to set values of parameters like parallelism and port number. The bash commands for Frontier can be seen in Appendix A. Initially, if the tool can be built to assess the performances of Apache Flink, Apache Heron, Frontier, NebulaStream and Timely Dataflow on an IoT infrastructure, then it will definitely create a lot of value especially for DevOps Engineers.

ACKNOWLEDGEMENTS

We would like to thank Eleni Tzirita Zacharatou and Ventura Del Monte from the Technical University of Berlin for supporting us as our mentors. Also, we are very grateful to Jorge-Arnulfo Quiané-Ruiz from the Technical University of Berlin for motivating us throughout the process of this study and giving us the opportunity to work on it.

REFERENCES

[1] IDC, 2019. Internet of things ecosystem and trends.
[2] Statista, 2020. Number of publicly known internet of things (iot) platforms worldwide from 2015 to 2019.
[3] IDC, 2019. The growth in connected iot devices is expected to generate 79.4zb of data in 2025, according to a new idc forecast.
[4] Steffen Zeuch, Ankit Chaudhary, Bonaventura Del Monte, Haralampos Gavriilidis, Dimitrios Giouroukis, Philipp M. Grulich, Sebastian Breß, Jonas Traub, Volker Markl, 2020. The nebulastream platform: Data and application management for the internet of things.
[5] Dan O'Keeffe, Theodoros Salonidis, Peter Pietzuch, 2018. Frontier: Resilient edge processing for internet of things.
[6] Adrian Bartnik, Bonaventura Del Monte, Tilmann Rabl and Volker Markl, 2019. On-the-fly reconfiguration of query plans for stateful stream processing engines.
[7] Jeyhun Karimov, Tilmann Rabl, Asterios Katsifodimos, Roman Samarev, Henri Heiskanen, Volker Markl, 2018. Benchmarking distributed stream data processing systems, April.
[8] CORE, 2004. coreemu.github.io/core/.
[9] Lab, U. N. R., 2004. Networks and communication systems branch.
[10] EMANE, 2004. coreemu.github.io/core/emane.html.
[11] Bartnik, A., 2018. github.com/adrianbartnik/masterthesisjobs.

A Appendix

The code snippets shown in this section are for Frontier and are included here to provide an insight into our contribution on Frontier. As a stream data processing system, it consists of a source, processor and sink. In the code snippet for Frontier Source, it can be observed that the window size has been set to 10 seconds and the window slide has been set to 1 second. The bash command to build the Frontier core needs to be executed in the root folder. The commands for the master node and worker nodes need to be run via the command line by double-clicking the respective

nodes (routers) in the EMANE topology. As mentioned in the commands, in this study we used the port numbers 3501,3502 and 3503 for the three worker nodes.

```
// Frontier - Source

      ArrayList<String> srcFields = new
          ArrayList<String>();
       srcFields.add(Constants.TIMESTAMP);
       srcFields.add(Constants.userId);
       srcFields.add(Constants.pageId);
       srcFields.add(Constants.ad_id);
       srcFields.add(Constants.ad_type);
       srcFields.add(Constants.event_type);
       srcFields.add(Constants.ip_address);
       Connectable src =
           QueryBuilder.newStatelessSource(new
           EventGenerator(eventsInSecond), 13,
           srcFields);

       int windowSizeMS = 10000;
       int windowSlideMS = 1000;
       StateWrapper sw = new StateWrapper(20,
           windowSlideMS);
       Connectable statefulOverStateless =
           QueryBuilder.newStatelessOperator(new
           StatefulWindowOverStateless(windowSizeMS,
           windowSlideMS, sw), 20, srcFields);

       // Declare sink
       ArrayList<String> snkFields = new
           ArrayList<String>();
       snkFields.add(Constants.TIMESTAMP);
       snkFields.add(Constants.EVENTS_IN_WINDOW);
       snkFields.add(Constants.LATENCY);
       Connectable snk =
           QueryBuilder.newStatelessSink(new Sink(),
           25, snkFields);

       /** Connect operators **/
       src.connectTo(statefulOverStateless, true, 0);
       statefulOverStateless.connectTo(snk, true, 0);
```

```
// Frontier - Processor

    public StatefulWindowOverStateless(int size, int
        slide, StateWrapper stateWrapper) {
       this.size = size;
       this.slide = slide;
       this.stateWrapper = stateWrapper;
    }

    public void processData(DataTuple data) {
       long messageTS =
           data.getLong(Constants.TIMESTAMP);
       String userId =
           data.getString(Constants.userId);
       String pageId =
           data.getString(Constants.pageId);
       String ad_id = data.getString(Constants.ad_id);
       String ad_type =
           data.getString(Constants.ad_type);
       String event_type =
           data.getString(Constants.event_type);
       String ip_address =
           data.getString(Constants.ip_address);

       payload++;
       Event newEvent = new Event(messageTS, userId,
           pageId, ad_id, ad_type, event_type,
           ip_address);
       replaceState(stateWrapper);
       currentWindow.add(newEvent);
       if (lastTriggerTime < messageTS - slide) {
          // send window
          DataTuple outputTuple =
              doWindowStatefullAggregation(currentWindow,
              data, messageTS, lastTriggerTime);
          api.send(outputTuple);

          currentWindow.removeIf(e -> e.timestamp <
              (messageTS - size));
          lastTriggerTime = messageTS;
       }
    }
```

```
// Frontier - Sink
public void processData(DataTuple dt) {
       long windowTS =
           dt.getLong(Constants.TIMESTAMP);
       int eventsCount =
           dt.getInt(Constants.EVENTS_IN_WINDOW);
       long payload = dt.getLong(Constants.LATENCY);
       long currTS = System.currentTimeMillis() -
           windowTS;

       if (System.currentTimeMillis() - latestUpdate
           >= 1000){
          System.out.println("output-window-lat," +
              windowTS + "," + currTS);
          latestUpdate = System.currentTimeMillis();
       }
    }
```

```
# Build the Frontier project in the root folder

./frontier-bld.sh core

# Run the master for stateful window query in Frontier

java -jar lib/seep-system-0.0.1-SNAPSHOT.jar Master
    `pwd`/dist/stateful-window-query.jar Base

# Run 3 worker nodes for stateful window query in
    Frontier

java -jar lib/seep-system-0.0.1-SNAPSHOT.jar Worker
    3501 2>&1 | grep output-window-lat

java -jar lib/seep-system-0.0.1-SNAPSHOT.jar Worker
    3502 2>&1 | grep output-window-lat

java -jar lib/seep-system-0.0.1-SNAPSHOT.jar Worker
    3503 2>&1 | grep output-window-lat



# Script to plot Frontier results

    last_generated_folder =
        os.path.join(SOURCE_DIR_PATH, folder_name)
    labels, latency = [], []
    for filename in os.listdir(last_generated_folder):
        if filename.startswith('.'):
            continue

        with open(os.path.join(last_generated_folder,
            filename), encoding="utf-8",
            errors="ignore") as fp:
            for row in csv.reader(fp, delimiter=","):
                ts = int(row[1])
                dt = datetime.fromtimestamp(ts // 1000)
                ms = ts % 1000
                label = (dt +
                    timedelta(milliseconds=ms))
                .strftime("%S")
                labels.append(label)
                latency.append(int(row[2]))

    labels.sort()
    labels = [str(l) for l in labels]

    return folder_name, labels, latency


# Script to print and plot system usage

def get_data():
    cpus = []
    memory = []
    timestamps = []
    for x in range(20):
        cpu, mem = get_process_info()
        ts = datetime.fromtimestamp(time.time())
        .strftime("%M:%S")
        cpus.append(cpu)
        memory.append(mem)
        timestamps.append(ts)
        time.sleep(1)
        print(cpu,mem,ts)

    return cpus,memory,timestamps


# Script to plot Apache Flink results

    last_generated_folder =
        os.path.join(SOURCE_DIR_PATH, folder_name)
    labels, latency = [], []
    for filename in os.listdir(last_generated_folder):
        if filename.startswith('.'):
            continue

        with open(os.path.join(last_generated_folder,
            filename),
        encoding="utf-8", errors="ignore") as fp:
            for row in csv.reader(fp, delimiter=","):
                labels.append(datetime
                .fromtimestamp(int(row[3]) // 1000))
                latency.append(int(row[2]))

    labels.sort()
    labels = [str(l) for l in labels]

    return folder_name, labels, latency
```

Report Authorship

Santosh Dhirwani:
Overview, Objectives, Implementation (Workload Design, Input Generator, Systems Under Test), Experimentation, Conclusions and Future Work, Acknowledgements, Appendix

Yiming Gu:
Abstract, Introduction, Overview, Objectives, Implementation (CORE, EMANE and Services)