Edge-Stream: a Stream Processing Approach for Distributed Applications on a Hierarchical Edge-computing System

Xiaoyang Wang*†, Zhe Zhou*†, Ping Han†, Tong Meng*, Guangyu Sun*†[1], Jidong Zhai‡
*Peking University, Beijing, China, {yaoer, zhou.zhe, mengtong, gsun}@pku.edu.cn
†Advanced Institute of Information Technology, Hangzhou, Zhejiang, China, hpdgy@163.com
‡Tsinghua University, Beijing, China, zhaijidong@tsinghua.edu.cn

2020 IEEE/ACM Symposium on Edge Computing (SEC) | 978-1-7281-5943-0/20/$31.00 ©2020 IEEE | DOI: 10.1109/SEC50012.2020.00009

***Abstract*—With the rapid growth of IoT devices, the traditional cloud computing scheme is inefficient for many IoT based applications, mainly due to network data flood, long latency, and privacy issues. To this end, the edge computing scheme is proposed to mitigate these problems. However, in an edge computing system, the application development becomes more complicated as it involves increasing levels of edge nodes. Although some efforts have been introduced, existing edge computing frameworks still have some limitations in various application scenarios. To overcome these limitations, we propose a new programming model called Edge-Stream. It is a simple and programmer-friendly model, which can cover typical scenarios in edge-computing. Besides, we address several new issues, such as data sharing and area awareness, in this model. We also implement a prototype of edge-computing framework based on the Edge-Stream model. A comprehensive evaluation is provided based on the prototype. Experimental results demonstrate the effectiveness of the model.**

***Index Terms*—edge computing; programming model;**

I. Introduction

Internet of Things (IoT) devices have experienced exponential growth in recent years. Applications based on IoT technology have been widely employed in various fields, such as healthcare, transportation, and manufacturing. Traditionally, data collected by those geo-distributed devices are processed centrally in a data-center. However, such a solution may flood the network with heavy data traffic due to the rapid increase of connected IoT devices. Also, when endpoint users are sensitive to the latency, it is neither efficient nor necessary to deliver all raw data into a remote data-center for computing. More importantly, many modern applications raise the requirements of data-location awareness and privacy protection, which cannot be satisfied by traditional cloud computing.

The concept of edge computing has been introduced as a promising scheme to solve these issues. The basic idea is to offload processing tasks close to those IoT devices, exploring the computing capability of nodes nearby. Furthermore, the advance of 5G communication technique involves the deployment of powerful servers with a normal operating system and adequate storage space throughout the network, such as macro-cell base stations, small-cell base stations, Cloudlet servers, and access points [26]. Therefore, the edge computing scheme is able to improve the utilization of those resources and avoid overwhelming the network with heavy data traffic. As illustrated in Figure 1, all the participants throughout the network are normally organized in a hierarchical structure. The data-center is located at the root, while those IoT devices form the leaves. The computing nodes in between comprise those internal nodes. All those nodes in the structure have varying degrees of computing and storage capabilities, which are rather considerable when accumulated together. Recently, extensive approaches of edge computing are proposed in the industry and academia [5] [20] [26]. They have well demonstrated the advantages of edge computing systems, in respect of latency, energy consumption, and privacy-preserving, etc.

However, in an edge computing system, the application development efficiency still remains as an critical issue. Compared to the two-tier cloud computing architecture, an edge computing framework is normally more complicated due to new participants (i.e. edge nodes). Therefore, it will be too sophisticated for developers to manage low-level parallelism and resource allocation details manually on such a group of multi-tier nodes. In order to make the edge computing scheme practical, a dedicated high-level programming model is necessary. It is supposed to be independent of the physical topology of the IoT network, and able to cover most of the common scenarios for various IoT applications.

Several projects have already noticed such demands, and focus on the convenience in application development by providing proper abstraction [27] [18] [12] [42]. They have explored a lot of techniques from various aspects, making full use of available resources to maximize their optimization targets. However, most of the existing approaches have several common limitations, which affect their efficiency in real deployment. They are explained in details as follows.

First, some frameworks focus on the case that there is only a single application running in the network. In multi-task cases, if the endpoint devices are grouped into isolated sets for each task, everything just works well as a single-task system. But, if they share the same set of data sources, each task will request the same data repeatedly, resulting in redundant data traffic. Some commercial products [3] [28] [14] [2] and community projects [10] [11] have tried to solve the problem by involving a message queue service over IoT devices. It actually provides

[1]Corresponding author

978-1-7281-5943-0/20/$31.00 ©2020 IEEE
DOI 10.1109/SEC50012.2020.00009

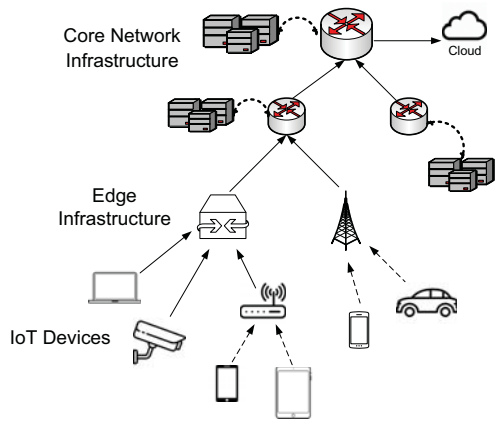


Fig. 1: Illustration of a typical topology of computing resources throughout the network

a publish-subscribe interface for different tasks. However, such type of abstraction is not friendly to developers. Users have to manage the data transmission details manually.

Second, data movement among endpoint participants is complicated in real-world applications. Most research approaches only focus on optimizations for some special cases, such as collecting data from IoT sensors towards the cloud, or reversely, broadcasting data content to endpoint devices. Real-world applications, however, may require more complicated data transmission paths.

Third, the collaboration among IoT owners has not been explored yet. IoT devices are normally deployed separately for different purposes. As device number keeps increasing, it is also possible that data collected by different devices are used together for new tasks. Thus, how to enable data sharing among users becomes a new challenge.

To overcome these limitations, we propose a new programming model called Edge-Stream. In this model, data flows are represented as streams so that low level details are hidden from programmers. In addition, the streams can be shared among different tasks. Various data movement patterns are supported in the model to cover common scenarios.

The rest of this paper is organized as follows. In Section II, we review the basics of edge computing and introduce four typical scenarios. In addition, the limitations of existing approaches are also discussed. To overcome these limitations, the Edge-Stream model is proposed in Section III. Based on this model, we further discuss several critical implementation issues in Section IV. Evaluation setup and results are presented in Section V, followed by a conclusion in Section VI.

II. Related Work

In this section, we first review the basic idea of edge computing and existing frameworks that support edge computing. Then, we study four typical scenarios of IoT applications that can leverage edge computing. After that, we summarize limitations of current approaches, which motivate the proposal of our Edge-Stream.

A. Edge Computing Basics

With the rapid growth in the number of IoT devices, both data and communication traffic generated by them have been increasing exponentially. Although modern cloud systems are elastic in computing and storage capacity, those embedded and mobile devices are normally located far from the data-center. Consequently, traditional cloud computing may suffer from severe round-trip delay and network jam. To mitigate this problem, researchers proposed the concept of edge computing. Its goal is to perform more data processing jobs at the edge of the network [5] [20], which is closer to IoT devices. Mahmud *et al.* [26] presents a detailed survey in this field. Both challenges and opportunities exist in all aspects [34].

To bring the idea into reality, all types of nodes close to the IoT devices are involved to share computation and storage resources. Cloud computing service providers such as Amazon AWS Greengrass [3], Microsoft Azure IoT Hub [28], and Google Cloud IoT Edge [14] present the most intuitive solutions. With the help of special hardware and software tool-kits, the IoT devices are able to perform some local inferences and manage the exchange of data with the nearest data-center. CloudPath [29] supports execution of third-party applications along a progression of data-centers positioned along the network path between the client devices and a traditional wide-area cloud data-center. Cloudlet [32] provides a small-scale "data-center" called Cloudlet servers located at the edge of the Internet to reduce the application latency. Early edge computing system Paradrop [39] leverages the potential computing capacity of access points, smart routers and other existing components of the network. Generally speaking, these infrastructures are geographically next to IoT devices, and are equipped with considerable computing and storage capability.

B. Typical Scenarios in Edge Computing

IoT devices have been applied in a huge number of different applications. These applications vary a lot, in respect of number of devices, hardware platforms, service types, communication infrastructures, etc. Thus, it is important to abstract some common scenarios from these applications to propose an efficient programming model. To this end, we introduce four typical abstractions of scenarios, as illustrated in Figure 2, according to data-flow. They can cover a lot of representative applications that can leverage edge computing.

The first scenario is called **IoT-Edge-Cloud**, as shown in Figure 2 (a). From the data-flow, we can find that the system gathers data from widespread geo-distributed IoT devices into several data-centers for processing. In this case, a full path data delivery from IoT devices towards the data-center is always necessary. Thus, some pre-computing on edge devices along the path can help reduce data size or overall latency significantly. Liu *et al.* [24] introduces the Markov Decision Process (MDP) method into the optimization algorithm, thus reducing the average delay of every task. Edgeflow [43] provides a method to smooth the data burst problem considering the constraints of the network. Flask [27] allows programmers

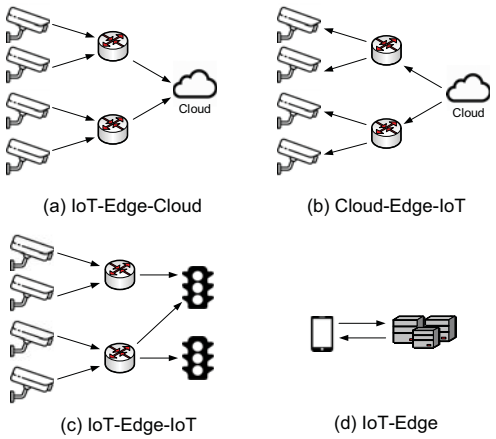


Fig. 2: Four typical scenarios of IoT applications

to describe their dataflow graph in a high-level programmatic wiring language on a sensor network, based on the functional language OCaml. Couper [19] focuses on deploying deep learning algorithms on a three-tier edge computing system. It supports quick creation of slices of production DNNs for visual analytics, and enables their deployment in contemporary container-based edge software stacks.

The second scenario is called **Cloud-Edge-IoT**, as illustrated in Figure 2 (b). It is common for applications like live-streaming. In this case, data are pushed down from data-centers towards a large group of endpoint devices. The key idea is to cache the hot content at the edge of the network, and thus reduce the transmission delay of contents and relieve the bandwidth pressure for data-centers. The content delivery network [36] has already become a mature approach for Internet-based caching by deploying cache servers at the edge of Internet. Information-centric networking [1] provides content distribution services to mobile users with a wireless cache infrastructure. Femtocaching [13] studies a wireless distributed caching network. It assists the macro base station by handling requests of popular files that have been cached. Xing et al. [40] proposes a distributed multi-level storage model to provide storage and computing services for IoT devices with limited storage capacity and poor network stability.

The third scenario is named as **IoT-Edge-IoT** in this work, as shown in Figure 2 (c). It refers to those applications that both data producers and consumers are endpoint IoT devices. When tracing the data-flow, we can find that data requests usually come from devices, which are close to the data source devices geographically. A straightforward method of edge processing for this scenario relies on an interconnected local network, such as the Wireless Sensor Networks (WSN) [30]. DDF [12] introduces an approach to develop programs using a dataflow model. This scenario is common for applications, such as smart transportation and smart manufacturing. In fact, Amazon AWS Greengrass [3], Microsoft Azure IoT Hub [28] and Google Cloud IoT Edge [14] have provided their architectures, where sensors collect data to analyze and actors react to the result. To the best of our knowledge, there are seldom frameworks that are able to provide range-based data aggregation. For example, EdgeX [10] and AWS Greengrass [3] allow users to create topics to collect data from sources and let sinks to subscribe. However, those topics are mostly defined in advance, instead of dynamically following the data sink. If a vehicle wants to know the number of cars around it, it is hard to select a proper topic to accumulate. A detailed description of the case is introduced in Subsection III-D.

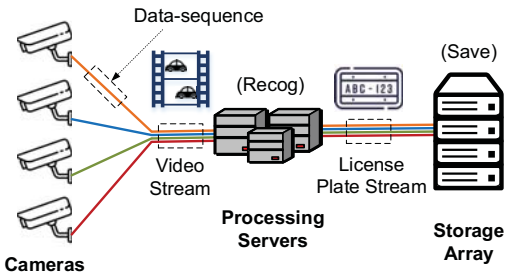
The last type called **IoT-Edge** focuses on a special scenario, in which data processing results are collected by the same data source device. Such case is common in mobile applications, such as gaming, VR, and AR. If the IoT or mobile devices are not able to complete the computation all by themselves, they have to borrow computing resources from other powerful nodes nearby. Cloudlet [32] allows devices to discover nearby Cloudlet servers to help them with the workload. Mobile Fog [18] proposes a Platform-as-a-Service (PaaS) model to provide developers with a simplified programming interface. Paradrop [39] serves the endpoint users with a multi-tenant platform. Chen *et al.* [8] demonstrates the multi-user computation offloading procedure as a game. The scheduling problem is transformed into reaching a Nash Equilibrium.

C. Motivation of Our Approach

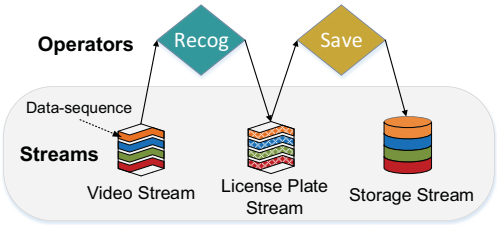
From the discussion in the last subsection, we can find that most edge-computing applications can be categorized into one aforementioned scenario or a combination of multiple ones, such as a complex smart city system [44]. Unfortunately, most state-of-art edge-computing systems/infrastructures only focus on a single scenario. There still lacks a unified programming model that can support all the scenarios. In addition, the low-level design details are not well hidden in some of edge-computing systems. Moreover, special hardware support is required for some of edge-computing infrastructures, such as Amazon AWS Greengrass [3], Microsoft Azure IoT Hub [28], and Google Cloud IoT Edge [14]. Thus, the development of edge computing on a real-world application is inefficient, especially when multiple scenarios are involved in one application.

Traditional distributed computation frameworks in data-centers, including Flink [6] and Storm [35], are designed to deal with continuous data packages generated from IoT devices, web applications and social media. The stream processing model is adopted to describe the jobs. The endless sequence of data packages of the same type are abstracted as a *stream*. Developers apply a series of *operators* to process each element in the stream. Such high-level abstraction is friendly to programmers, and is expressive to describe different types of jobs in data-centers.

Inspired by the method, we introduce a new stream processing model, Edge-Stream, for edge computing scenarios. It is rather simple and easy to program, but is general enough to cover the typical scenarios mentioned above. At the same time, we implement a prototype of edge-computing framework based on Edge-Stream model on commodity hardware. Details are introduced in following sections.



(a) Physical system illustration



(b) Abstract view of the job

Fig. 3: Different views for a license plate recognition job

III. Edge-Stream Model

In this section, we mainly introduce the components of the Edge-Stream model. First, we provide a case study in order to facilitate the explanation. Then we describe the stream abstraction from the perspective of each basic components. As the Edge-Stream model borrows some concepts from the traditional file system, we also compare some coincident part between them, and emphasize the specific features for streams.

A. Basic Components

When we use Edge-Stream model to describe a scenario, it contains several basic components, including "data-sequence", "stream", and "operator". Their definitions are introduced as follows using a simple example of intelligent traffic system illustrated in Figure 3.

a) Data-sequence: Data-sequence is the basic unit that can be manipulated by an operator in the model. As shown in Figure 3 (a), each camera generates a data-sequence of captured vehicles. These data-sequences are used for data analysis (e.g. license plate recognition) later in the system.

b) Stream: Stream is a set of data-sequences, which are processed with the same operator. As shown in Figure 3 (a), there are four cameras in total. All data-sequences from these cameras are grouped together as a video stream.

c) Operator: Operator is a user-defined function in an edge system. It takes one or multiple streams as the input and generates a single stream as the output. For the example in Figure 3 (a), there are two operators. The first operator is *License Plate Recognition (Recog)*. It takes the video stream from cameras as input and generates a license plate stream, which contains the license plates of vehicles. The second operator is *Save*. It takes the license plate stream as the input and turns it into a new stream for storage.

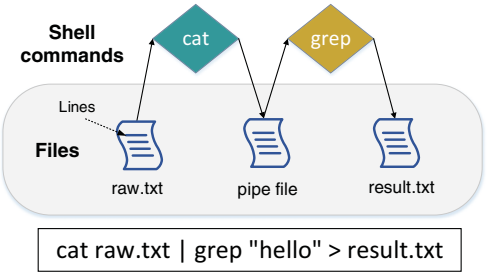


Fig. 4: An analogy with Linux shell script example

The abstract view of this example is illustrated in Figure 3 (b). It contains three streams and two operators. In each abstract view, we put operators on the top and allocate all streams in the bottom. In the rest of this paper, we will use abstract views to introduce more examples. The corresponding code based on Edge-Stream model is presented in Listing 1. The first line declares two operators (i.e. Recognition and Save). The second line imports the video stream. License plate stream and storage stream are generated using operators in last two lines, respectively.

```
require Recog, Save
import VideoStream
PlateStream = Recog VideoStream
StorageStream = Save PlateStream
```

Listing 1: Code for the license plate recognition job

To provide better understanding of these components in Edge-Stream model, we use the file system as a close analogy. A stream in an edge system can be analogous to a file in a file system. A data-sequence can be analogous to a line in each file. The operators are analogous to shell commands, such as *tail, grep, cat*.

We also provide an example of manipulating files via Linux shell commands, which is shown in Figure 4, to demonstrate the similarity. Command `cat` reads each line in raw.txt, and pass on them with a pipe. The pipe process will create a new pipe file in the PipeFS, a special file system mounted in the kernel. Then command `grep` reads each line in the pipe file, and writes the result to result.txt. It is easy to tell that the example of manipulating file in Figure 4 is quite similar to that of programming with Edge-Stream model in Figure 3 (b). Design details of "file (stream)" and "shell commands (operators)" in Edge-Stream model are introduced in the following subsections.

B. Stream Design

Besides a set of data-sequences, a stream also maintains a list of metadata indicating its properties. Operators need to check those metadata items to determine whether the stream meets the requirements as an input. The key items of the stream metadata are listed in Table I. Other minor features, such as the stream name, are omitted from the list.

TABLE I: Basic metadata items of a stream

Metadata	Meaning
Type	Classifying the stream into several categories: primitive, virtual, generated.
Owner	Indicating the user that creates and possesses the stream.
Window	Describing the window type of the stream.
Serializer	Indicating the serializing method.

According to the `type` field in metadata, streams are categorized into three types, which are introduced as follows.

- A `primitive` stream is generated directly from end-point physical devices. Examples include streams generated by end-point sensors, surveillance cameras, streaming media from the cloud, and so on. Therefore, the video stream in Figure 5 (a) is a primitive stream.
- A `virtual` stream introduces another type of data source. Different from a primitive stream, it does not rely on any physical devices. Each node in an edge system may maintain a local replication of an identical virtual steam. Therefore, it is no necessary to move a virtual stream among nodes. A timer stream shown in Figure 5 (b) is a typical virtual stream, which provides the current time every minute.
- A `generated` stream is the output generated an operator, as illustrated in Figure 5 (c). In other words, it always relies on existing streams (i.e. input of an operator). We call those streams as the "parent streams" of the generated stream. For example, the license plate stream mentioned in Figure 3 is categorized as a generated stream, and the video stream is its parent stream.

Primitive streams are the original sources of data in the system. There are basically two kinds of them: device-based and range-based. They get data-sequences from a list of physical devices, and a list of areas, respectively. For example, stream owner X has deployed a bunch of sensors, and is able to get data-sequence from them. Thus it is device-based stream. An other case is that stream owner Y wants to provide services for vehicles but does not possess vehicles. Then Y may create a range-based primitive stream according to a range of areas. Any vehicle that enters the area becomes a data source of the stream automatically, and thus is able to access Y's service. More implementation details are introduced in Section IV.

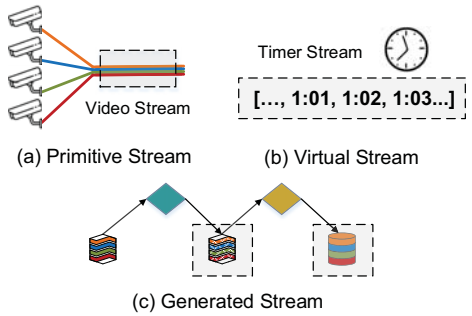


Fig. 5: Different types of streams

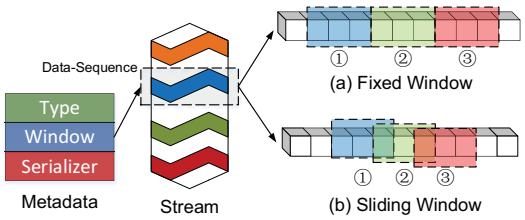


Fig. 6: Two typical types of windows

The following code 2 provides a simplified declaration of a video stream and a timer stream mentioned in Figure 5. The video stream is a device-based primitive stream, whose declaration contains a list of devices. When the stream is in use, the system will search for those devices and inlet the data. A virtual stream declaration needs to define the method of generating the virtual data. Once activated, they are generated locally by the required nodes in the system. In contrast, generated streams are defined via operators, such as the PlateStream in code 1.

```
<Stream>
  <Type>Primitive</Type>
  <SourcePool>
    <Source>192.168.10.*</Source>
  </SourcePool>
</Stream>

<Stream>
  <Type>Virtual</Type>
  <SourcePool>
    <Source>date +%M%S</Source>
  </SourcePool>
</Stream>
```

Listing 2: Stream declaration examples

Since the aggregation on an endless stream would never return, the concept of `window` is widely accepted in traditional stream computing frameworks, such as Flink [6] and Beam [4]. It indicates the range of data to be aggregated. In Edge-Stream model, the usage of `windows` remains the same. Figure 6 illustrates a stream with four data-sequences, and we focus on one of them. A fixed window divides up the sequence into fixed-width and non-overlapping time intervals. A sliding window only considers the starting point, and allows overlapping. For example, each window in Figure 6 (b) chunks 3 seconds of data, but a new window starts every 2 seconds.

The `Serializer` field indicates the serializing method of data-sequences in a stream. Given a serializer, operators are able to decode the input data and access the content properly. Common approaches include JSON, Protobuf, and Kyro.

In order to clarify the usage of a window, we take the vehicle counting job as an example. Users want to figure out the number of cars captured by each camera every hour. They will reuse the PlateStream generated from code 1. As shown in code listing 3, **FWindow** generates a new stream with an

one-hour window from the PlateStream. Notice that **FWindow** is followed by a pair of parenthesis for non-stream parameters (i.e. "1h"). It is then piped to '**VCnt**' operator to perform the final counting. Obvioulsy, the usage of a pipe is also similar to that in a file system.

```
require FWindow, VCnt, Save
import PlateStream
CountingResult =
  FWindow("1h") PlateStream | VCnt
StoreResult = Save CountingResult
```

Listing 3: Code for the vehicle counting job

C. Operator Design

In the Edge-Stream model, there are two types of operators, namely reshaping and computation. *Reshaping operators* are introduced to define how to organize existing data-sequences, without changing the data inside. For example, a `Union` operator collects all the data-sequences from each of the input streams, and treat them as a new stream. The `FWindow` operator in code 3 also belongs to this type. It only changes how the system treats the data-sequences. Such concept is similar to the *string_view* method in C++ language.

Computation operators, on the contrary, generate new data from input streams. They apply functions on each of the data-sequences in the input streams. Formally, an operator with function f operates on a stream with three data-sequences $\{a, b, c\}$ is demonstrated in Equation 1. Therefore, functions are the pivots of designing an operator.

$$Operator_f(\{a,b,c\}) = \{f(a), f(b), f(c)\} \quad (1)$$

Functions access data packages in each data-sequence through a standard set of APIs. Two of the most important APIs are `getNext` and `getWindow`. They correspond to two different types of functions, `map` and `reduce`, respectively. The `map` function takes in one data-sequence from a stream and uses `getNext` to access each data item in the data-sequence. Then, it generates a new data-sequence for the output stream. The `reduce` function involves `getWindow` to iterate data packages in each window of the data-sequence. It waits for all data in a window to accumulate in a physical node to start processing. They are the basic components adopted by most of the distributed systems [9] [7] [6]. Those APIs do not limit the data type of the operator, which support heterogeneous data sources. The data type of the data packages in the data-sequence is declared implicitly in the operator, demonstrated in following examples (listing 4 and 5).

The code snippet in listing 4 provides a simplified implementation of the function for the **Recog** operator in Figure 3. It is a map function. The code starts with an initialization section in C++ language. The following declaration part describes the input and output streams. In the example, there is one input stream `S_video` consisted of `Picture` objects. The output stream `S_plate` provides `std::string` objects. The last section implements the computation logic with C++ code. The `getNext` function in the first line pops one element at a time from the stream. The '%%', '%{' and '%}' are punctuation to separate the sections.

```
#include <string>
#include "MyRecogLib"
%%
%in S_video<Picture, null, File>
%out S_plate<std::string, null, JSON>
%%
%{
  auto inPicture = S_video.getNext();
  auto outPlate = PlateRecog(inPicture);
  S_plate.pushItem(outPlate);
%}
```

Listing 4: Recog function implementation

The code listing 5 implements the vehicle counting operator. The input stream `S_plate` has been segmented with a fixed window. The output stream `S_result` provides the counting result without a window. Both of them are serialized in JSON. The function intends to calculate the total number of plates in each window. The `getWindow` function performs in a `reduce`-style manner. It collects data in each window and returns a vector of elements.

```
#include <string>
%%
%in S_plate<std::string, fixed, JSON>
%out S_result<int, null, JSON>
%%
%{
  int counter = 0;
  auto plates = S_plate.getWindow();
  for (plate : plates) {
    counter ++;
  }
  S_result.pushItem(counter);
%}
```

Listing 5: Vehicle counting function implementation

The Edge-Stream model also supports operators to maintain **per-data-sequence status** for its output stream. For example, users want to know the total number of cars from now on. They should replace the local variable `counter` with a global one following a status declaration:

```
%status counter<int, 0>
```

D. Grouping Method

Typical data processing frameworks in data-centers always support grouping methods. For example, the `keyBy` transformation in Flink [6] logically divides a stream into disjoint partitions. Records with the same key are assigned to the

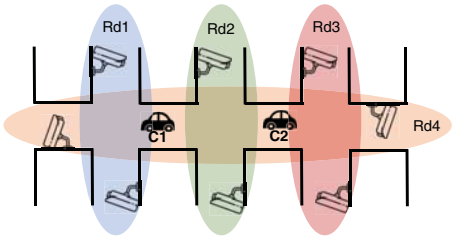


Fig. 7: Grouping example

same partition. To support various IoT applications, similar operations are also necessary in the model. In some cases, users have to recollect the data-streams in a stream on demand.

Consider an example of the IoT-Edge-IoT scenario. A smart traffic system provides information for each vehicle about its nearby traffic conditions. The user has already got a license plate stream from an existing job shown in Figure 3. Code 6 firstly accumulate the number of cars in each area. Then the results are delivered to nearby vehicles. The **LeftUnion** operator unions data-sequences of the second parameter according to the first one. Three key features, namely `areas`, `keys` and `rules`, are involved in such procedure.

```
require LeftUnion, Sum, Deliver
import CountingResult, AreaSet, Vehicles
TrafficPerArea =
  LeftUnion AreaSet CountingResult | Sum
SmartTraffic =
  LeftUnion Vehicles TrafficPerArea |
      Deliver
```

Listing 6: Code for the smart traffic job

Literally, `area` indicates where the data-sequence comes from. By default, the feature for a data sequence generated by an IoT device is set to the position of it. For generated operators, the `area` feature for the output data-sequence is assigned as the union of all `areas` of inputs. Primitive streams from clouds and virtual streams have no `area` feature. The `key` feature is a user-defined string assigned to a data-sequence. It often tells what the data-sequence is.

The `rule` feature of each data-sequence tells its listening range. They are set explicitly by stream owners. Only data-sequences that follow the `rule` will be forwarded to it. In the Edge-Stream model, there are two steps to define a `rule`. First, a `range` parameter is necessary for all `rules`. A positive `range` refers to a circle centered on the geometric center of its *area*. Its value is the radius measured in meters. The `range` with value '0' means the data-sequence itself. It is always used in the IoT-Edge scenario. A negative number represents that all available data-sequences are accepted. Second, a series of `keys` are set to further select data-sequences.

Figure 7 provides an illustration about the traffic system. There are four roads, Rd1˜Rd4, in the system. Each road has two cameras running vehicle counting jobs. C1 and C2 are two cars running on road Rd4. Both of them want to figure out the nearby traffic condition. As shown in Code 6, the first step is to aggregate the counting result on each road. The AreaSet indicates the mapping from cameras to roads. Then, cars select the traffic condition data. Each of them defines a rule according to its location. As a result, C1 and C2 receive the data from {Rd1, Rd2, Rd4} and {Rd2, Rd3, Rd4} respectively.

E. Stream Sharing

Similar to a file system, Edge-Stream model also adopts the idea of permission control. In a typical file system, file permissions determine the ability of the users to access the file. Such mechanism enables multiple users to collaborate in a single system properly.

In Edge-Stream model, each stream has a unique owner. It refers to the user that creates and possesses the stream. The owner has full permissions to generate new streams from it, modify its metadata and delete the stream. For example, the primitive stream in code 2 has data sources at IP addresses *192.168.10.**. If the owner deploys a new devices at *192.168.1.100*, a corresponding source item will be added in the *SourcePool* list.

Owners can also share streams with other users. For example, there are two users, *A* and *B*, in Figure 8. User *B* attempts to build a license plate recognition job mentioned in Figure 3. However, *B* does not have enough camera devices, and thus asks *A* to share a stream to fertilize the data sources.

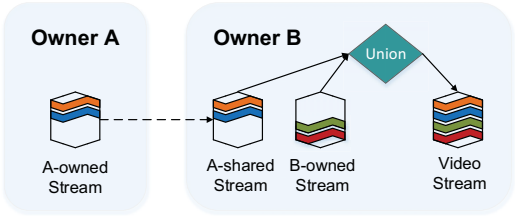


Fig. 8: A stream sharing example

When user *A* starts to share the stream with user *B*, *B* will get a reference of it. Then, *B* is allowed to build new streams from it, but cannot modify or delete the stream in the edge system. The metadata of the stream managed by *A* should also record the event, in order to close the sharing afterwards. With the help of stream sharing, existing streams are not only reused by creators to build their own jobs, but also shared among different users. Such progress is similar to creating soft links of files. The file content appears only once on the disk, but has multiple entries in the system.

IV. EStream Platform

Based on the Edge-Stream model, we implement a full-stack prototype named EStream. In this section, we first discuss the challenges that the system should take into consideration. Then we will provide an overview of the EStream architecture, as well as some low-level design concepts to implement streams in the system. A case study will be presented in the following subsection. We address the decentralized scheduling algorithm design in the last subsection.

A. Challenges

Stream processing is a typical form of computation jobs in data-centers. There have been lots of famous frameworks in use for years, such as Flink [6] and Storm [35]. AStream [22] provides a practical way to integrate ad-hoc queries with long-running queries on streams, which enables the sharing of the computation. However, those models cannot be applied on the edge computing scenario, due to the following reasons.

First, stream processing frameworks in data-centers are built on clusters that are connected with each other via a high-speed wired network infrastructure. Data access latency is nearly independent of the physical location of the computation nodes. In comparison, computation nodes in edge computing systems are no longer connected as a complete graph. For a particular application, the topology of nodes throughout the network normally forms a level structure, as shown in Figure 1. The data-center is situated at the root logically. Those IoT devices make up the leaves. Computation nodes in between plays the part of internal nodes. Given the physical schema, we should figure out how to map the computing paradigm onward. In other words, determinate how streams are *created* and *go through* those nodes. It is the most important challenge to bring the Edge-Stream model into reality.

Second, the master-worker architecture of those frameworks in data-centers becomes inefficient, as the computation network is always larger in edge computing systems. The job placement becomes more difficult because of the large number of computation nodes. Furthermore, the communication overhead among computation nodes is significant. Thus, it is difficult to adjust the scheduling policy timely according to real-time changes in the remote edge network. If there are multiple users sharing data streams, it is also hard to select the maintainer of master nodes. Traditional resource management algorithms, such as Yarn [38], Mesos [17] and Omega [33], cannot work properly in this scenario. Therefore, a practical *scheduling method* is necessary for the EStream platform.

Third, we argue that the sharing of streams becomes more and more important for edge computing applications. In this scheme, multiple jobs are able to share their common parts of the computation graphs, and thus reduce the replicated computation and data transmission overhead in the system. Traditional stream processing frameworks cannot share a part of the existing job dynamically, because their scheduling unit is a job instead of a stream. In order to achieve the goal, they should stop the existing job, modify the computation graph, and restart the new one. This method is acceptable in data-centers. But, the cost becomes larger in edge computing systems with a huge number of remote nodes. Thus, it is also significant to provide a practical way to support the *stream sharing* feature in the EStream system.

In the following subsections, we will discuss how to solve those problems.

B. System Overview

In this subsection, we provide an overview of the EStream architecture, and describe how streams are *created* in it.

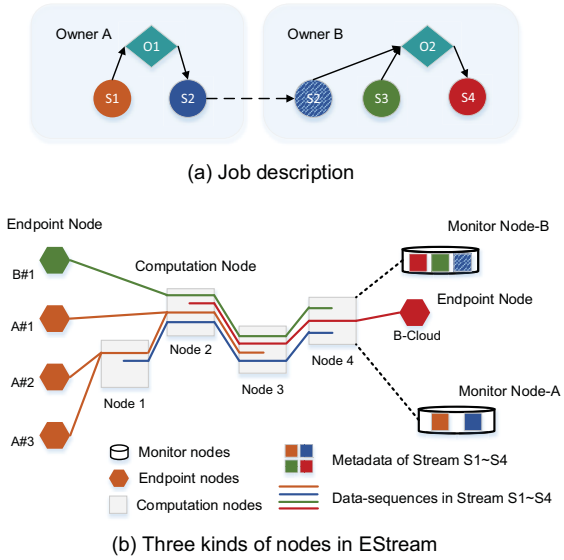


Fig. 9: Streams passing by EStream nodes

There are basically three kinds of devices in the EStream system: endpoint nodes that produce data-sequences, computation nodes that provide computation and storage resources, and monitor nodes that maintain the metadata of streams. Figure 9 illustrates an example for them. Part (a) describes the components of jobs for two owners, A and B. Owner A imposes an operator O1 on the stream S1. The result stream S2 is then shared to B, together with S3, as the input of O2. The output of B's job is S4. Part (b) illustrates the overall system topology for those nodes and streams.

The `endpoint nodes` include IoT devices and data-centers in the cloud. They provide the source of primitive streams in the system. In comparison, virtual streams come from device-independent algorithms, and generated streams are outputs of operators that take existing streams as inputs. They always start from `computation nodes`. Four hexagons on the left in Figure 9 (b) are endpoint nodes. The green one belongs to Owner B for stream S3. The orange ones belong to Owner A, generating three data-sequences for S1. The red hexagon on the right side, B-Cloud, shows another endpoint node of Owner B. It is the data sink of stream S4. Gray squares in the middle represent four computation nodes in the system. Lines passing through them illustrate the data-sequences in each stream. As shown in the figure, the primitive stream S1 starts from endpoint nodes A#1~#3, while a generated stream S2 (the blue line) starts from Node 1.

Besides endpoint and computation nodes, EStream incorporates some special `monitor nodes` to maintain the necessary information streams. They provide services to interact with streams, such as creating, deleting, and sharing. In Figure 9 (b), there are two monitor nodes, belonging to Owner A and B respectively. Monitor Node-A maintains the information of S1 and S2, while Monitor Node-B maintains S3 and S4, as well as the shared stream S2.

C. Stream Creation

Once a stream is created, it is registered in a monitor node provided by its owner. For primitive streams, the monitor maintains a list of endpoint devices for device-based streams, or a list of areas for range-based ones. Neither data transmission nor computation is triggered in this stage. Similarly, as virtual streams are built on demand, the monitor node only records the algorithm to generate them.

In comparison, creating generated streams is more complicated, as illustrated in Figure 10. In the example, Stream S has one parent stream named T. Initially, the monitor Node-B maintains the metadata of Stream T. ① After the registration procedure, Node-A has to ask Node-B for Stream T. ② Then Node-B performs an authentication verification about the event. If the verification is successful, it will help to figure out the current location of T. In the meantime, Node-A registers a shared stream T locally. In return, Node-B also records the information of Node-A, so as to end the sharing in the future. ③ At last, EStream triggers the data transmission and computation for S. In this way, the same primitive or generated stream in different jobs appears only once in the physical system. Therefore, this mechanism realizes the feature of *stream sharing* among owners.

According to the procedure, when creating streams from existing ones, their monitors are supposed to locate them in the system. There are three types of streams, which are primitive streams, virtual streams, and generated streams. (1) To find a primitive stream, the most intuitive way is to access the list of endpoint nodes maintained in the monitor node. A better way is to make use of the topology illustrated in Figure 1 to minimize the communication overhead. The basic idea is to treat the monitor node of a stream as the root, and endpoint nodes make up the leaves. The monitor node is able to reach all leaves through a breadth-first search on the graph. To support the searching procedure, internal computation nodes caches the list of neighbors to search. For example, in Figure 10, Node *2* maintains a list of $\{4, 5\}$ as the children set for the breadth-first search of Stream T, and Node *1* will make the list as $\{2, 4\}$ for Stream S. (2) Virtual streams are created on demand, and thus their monitor nodes do not have to find existing ones in

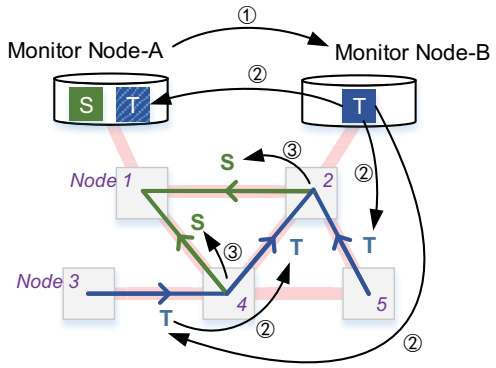


Fig. 10: Create a generated stream S from stream T

the system. (3) Generated streams are constructed on data-sequences from existing streams. In order to find a generated stream, its monitor node recursively finds all its parent streams in the system. Once all the original data-sources are found, it is easy to find the target stream following the direction of data transmission.

For example, Stream S in Figure 10 is a generated stream with one parent stream named T. In order to locate Stream S in the system, the monitor Node-A has to ask Node-B for nodes that provide data for Stream T (Node *2, 3, 4, 5*). Then the searching procedure starts from those nodes. Following the data streams, it is easy to find the offspring nodes *1, 2, 4* that generates data for Stream S.

D. Request Propagation

The previous subsection answers what happens when owners create new streams. However, for any pairs of nodes in the system, there are lots of possible routing paths for data transmission. We should clarify how data-sequences in those streams choose their way to *go through* the computation nodes in the system. The problem is divided into two parts: the direction for each data-sequence to go, and the location where it is generated.

First of all, EStream defines a target node for each data-sequence in streams. The data-sequence ought to find the shortest way towards its target. In other words, the target is the **direction** to deliver the data-sequence. Some of the streams are created to be sinked to somewhere, such as the data-center for StoreResult in Code 3, the endpoint vehicles for SmartTraffic in Code 6 and B-cloud for S4 in Figure 9. Those nodes are just the targets for data-sequences inside the streams. Other streams have no intuitive destinations, such as the temporary stream TrafficPerArea in Code 6 and S2 in Figure 9. For data-sequences in such kind of streams, the target nodes are set as the monitor node. As we introduced previously, each stream has only one monitor node in the cloud, so the target is still unique. If a stream is shared to another owner, its target nodes will not get changed. For example, although T in Figure 10 is shared from monitor Node-B to Node-A, its target node is still Monitor Node-A.

Besides the direction, it is also important to decide on where each data-sequence **first appear**s in the system. Obviously, data-sequences in primitive streams are launched from particular endpoint nodes. In virtual streams, they only appear in computation nodes on demand. For generated streams, the question is equivalent to determining *where the computation starts*, as those streams are generated from operators.

For computation operators, the input data-sequences are processed as soon as possible. The difference between `map`-style and `reduce`-style functions is that the latter one should wait for data in a window to accumulate. For reshaping operators, there are two possible cases. If the operator does not involve any grouping method, it only affects how users treat those data-sequences. So the underlying system will ignore it. For example, the `Union` operator unifies two or more streams into one. It only leaves a record in the metadata maintained

Algorithm 1: Pseudo code for AreaFinding

```
Input: Area, Node, Stream
Output: The node to collect data-sequences
if Area ⊄ Node[Stream].area then
    return Node[Stream].parent
end
foreach node ∈ Node[Stream].children do
    Result = AreaFinding(Area, node, Stream)
    if Result ≠ Node then
        return Result
    end
end
return Node
```

by the stream owner, without changing the existing data-sequences. In comparison, grouping operators are involved to collect a bunch of input data-sequences together into one physical node. In this case, EStream uses the *nearest common ancestor node* of their target nodes to collect them. Therefore, the key point of such procedure is to select the proper data-sequences according to the `rule` feature. As introduced in Subsection III-D, each `rule` consists of a `range` parameter and a series of `keys`. From the perspective of EStream system, the `range` parameter delimits a list of `areas`. Input data-sequences in each `area` are collected together.

In order to find matching `areas` quickly, EStream system introduces the `area` feature for those internal computation nodes as well. The area and related data structures are per-stream information. The mechanism reuses the parent-children relationship in the per-stream tree structure defined in Subsection IV-C. For a device-based primitive stream, the parent node of the endpoint node collects the positions of them, and calculates a smallest convex polygon to cover all of them as its own area. Recursively, the parent of the computation nodes records its area as the smallest convex k-gon to cover all of its children. In order to control the space overhead of the auxiliary data, we limit the polygon to at most m sides. Therefore, the space overhead for the area information in each computation node is at most O(m*N), where N denotes the number of streams. The idea is similar to the interval tree, which is a popular data structure to find all intervals that overlap with any given interval or point.

To find the matching `area` in the system, the target node will firstly ask all nodes connected to it. If the `area` of this node is able to cover, it will propagate the request among its children. Otherwise, it will ask its parent for help. The procedure continues recursively, until the system is able to find the proper node with the smallest `area`. Notice that the `area` refers to the smallest convex polygon that is able to cover all the `areas` of its children. Therefore, this node has the property that all data sources of data-sequences following the rule come from its children. EStream just use this node to start the collection job. The whole procedure is shown in Algorithm 1.

E. Case study

In this subsection, we provide a simplified case study, following stream owners A and B in Figure 9, and see what happens when they publish their jobs step by step. We select several physical nodes from Figure 9, providing a detailed illustration in Figure 11. It shows how data-sequences and operators are organized in those nodes. Circles and diamonds stand for streams and operators respectively. Squares in gray represent several endpoint and computation nodes.

Initially, there is no streams for both A and B. Each computation node will set its area as an empty set, indicating that it has not yet connected with any devices. When owner A and B publish primitive streams with endpoint IoT devices A#1~A#3 and B#1, those sensors are registered in primitive stream objects S1 and S3 respectively. Each IoT devices will report its parent computation node about its area. Then the computation node will calculate the smallest polygon to cover all those areas and use it as its current area. Recursively, the parent nodes of computation nodes will also update their area following the same method. That is, for Node 1, its area should cover both A#2 and A#3, while Node 2 should cover both A#1, B#1 and Node 1. Until then, no data packages generated by endpoint devices are transmitted yet, as there are no generated streams that need the data.

Then A creates a new stream S2 with operator O1. As S1 is its parent stream, the monitor has to find the location of S1 first. In Figure 9 (b), the searching procedure begins at Node 4 and ends at Node A#1~3. Both of the nearest computation nodes, Node 1 and Node 2, will register S1 and S2 streams inside, and link them with O1. Still, the calculation hasn't happened yet. After that, A shares S2 to B. According to the discussion previously, nothing happens in those computation nodes, but B is now able to access the stream S2. At last, B builds the stream S4 with operator O2, which intakes both S2 and S3 as its inputs. Assume that O2 is an operator with side effects, such as dumping the result to a database. Therefore, data transmission and computation in each nodes are triggered.

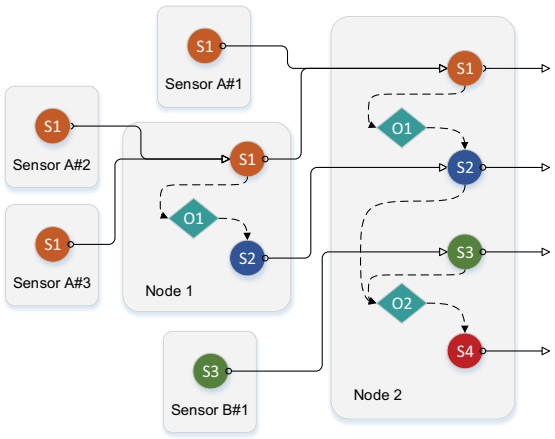


Fig. 11: Operators in computation nodes

F. Decentralized Scheduling

We have already known which nodes to perform the operators, but how to divide the workload among them still remains unclear. This requires some practical *scheduling algorithms*.

As we analyzed in Section IV-A, scheduling with global master nodes is inefficient in large-scale edge computing scenarios. Edge computing systems based on cloud management platforms, such as KubeEdge [41], will meet the scalability problem when the scale of computation network grows. Therefore, EStream choose to implement a decentralized method to allocate the computation workload in the system.

In stream processing scenarios, it is rather common that data accumulate in some nodes unexpectedly. There are many reasons: abrupt data bursts, unstable network conditions, garbage collections in computation nodes, etc. A possible way is to tell the previous node to slow down the data transmission rate, and thus pushes the pressure backwards to the data source. Such technique is named back-pressure, which has been adopted by Flink [6], Storm [35] and many other famous frameworks in data-centers. It occurs that the back-pressure finally reaches the original data sources. For data-center applications, they are reliable data stores such as Kafka [23]. In edge computing applications, however, the end-point data sources are always naïve sensors without capability to store lots of data. Therefore, the scheduling algorithm in EStream should choose another direction with abundant computation and storage resources to push the pressure.

Figure 11 illustrates the job deployment procedure among several nodes. The data-sequences in S1 come from Sensor A#1˜A#3, and processed by operator O1. A key observation is that both of Node 1 and 2 have arrows pointing out of the node from S1. They indicate that if O1 is not able to finish all the computation in the current node, the rest of the input data are forwarded to the next node. The *next* here refers to the following node on the way to the target nodes of data-sequences in S2. The scheduler adjusts the workload for each node by deciding on the ratio of packages that should be passed to the next node.

Based on the observation, we adopt a rather simple but effective scheduling method in our system. All the examples mentioned in this paragraph refers to the job illustrated in Figure 11. Briefly, the scheduling algorithm aims to adjust the workload among nodes until packages in the same data-sequence spend the same amount of time in each node. For example, O1 converts data packages in S1 into S2. The lifetime of packages in S1 includes the total transmission time, the queuing time in computation nodes and the final computation time. We denote the average lifetime of packages in Stream i on Node j as L_i^j. The algorithm tries to balance the value of L_{S1}^1, L_{S1}^2 and L_{S1}^3 by workload redistribution.

There are several efforts to achieve the goal: First, the algorithm selects to compute data packages with largest transmission latency in the same stream. For example, if Node 2 receives two data packages from Sensor A#1 and A#2 at the same time, the latter one is more likely to be executed, because it is a two-hop package. Second, the algorithm prefers to push the pressure towards data sources. If $L_{S1}^2 > L_{S1}^1$ and $L_{S1}^2 > L_{S1}^3$, the algorithm tends to make Node 1, instead of Node 3, compute more. In this way, the data source gains "attraction" to the computation of the workload. Third, the generated stream reduces the pressure of its parent streams, and vice versa. For example, in Node 1, assume the basic lifetime of packages in S1 and S2 are l_{S1}^1 and l_{S2}^1. Then both of L_{S1}^1 and L_{S2}^1 are computed as $min\left\{l_{S1}^1, l_{S2}^1\right\}$. It promotes the fusion of computation, and thus improves the temporal locality of data packages. Notice that data sinks also gains "attraction" to the workload, because the lifetime of those packages lasts until they reaches the sink. Both data sources and sinks pull the computation towards its own side, so the length of path is reduced, just like a tug-of-war.

In short, the basic idea of the method is to trigger competitions among nodes, until an approximate balance reaches. It is a decentralized algorithm without any master nodes, and avoids the back-pressure problem through the lifetime-based workload pressure adjustment. Notice that it is just one of the possible scheduling methods. The EStream framework does not limit the implementation of other kinds of scheduling algorithms for it.

V. Evaluation

In order to evaluate the benefits of EStream and demonstrate its superiority against baseline systems, we conduct comprehensive evaluations. Specifically, we first select a representative application for edge computing paradigm, namely smart traffic as our test case. Then we implement both the EStream system and baseline systems using the popular OMNet++ simulator [37]. To make the estimation more accurate, we also generate some profiling data, including various latency and energy consumption of jobs on real hardware platforms. Then we inject these profiling data to the simulator as the configuration parameters. The details of our evaluation methodology are introduced as follows.

A. Test Case

For evaluation, we take the smart traffic system to serve as the test case. Smart traffic system is considered as a cornerstone of a smart city [16] [21]. In a typical smart traffic system, there are usually IoT devices, edge servers and cloud servers. In our test case, the IoT devices can be surveillance cameras that are deployed along some roads to capture the pictures of vehicles and send them to edge servers for further analysing. Besides, the vehicles are also viewed as IoT devices, they receive messages such as the congestion condition from nearby edge servers. Edge servers are deployed to collect, process and distribute data. They interact with near by IoT devices through low-latency LAN and communicate with remote cloud servers through a high-latency core network. We introduce three typical jobs in the smart traffic system:

Job x: Vehicle detection. Vehicle detection is a most basic job for smart traffic system. The edge servers are used to detect all of the vehicles within live video frames uploaded by

the connected surveillance cameras. Basing on these detected vehicle pictures, we can further conduct several higher-level analysing. For example, we are able to count the amount of vehicles to predict traffic congestion, or recognize license plate numbers of the detected vehicles to locate some wanted vehicles. We refer vehicle detection as `job x`.

Job a: Licence plate numbers recognition. Recognizing Licence plate number is the most straightforward way to identify a vehicle. We set `job a` to recognize the licence plate numbers based on captured vehicle pictures. The recognized plate numbers are then uploaded to the cloud server for further analysing, such as comparing them with some wanted hit-and-run vehicles.

Job b: Vehicle attributes recognition. Besides licence plate, the vehicle attributes, such as color, logo, model and even subtle inter-class difference [45] are also of great importance to help do identification and retrieval. Note that in our case, `job b` is considered as a emergent task, which is requested less frequently than `job a`.

B. Experimental setup

1) Network Topology: The topology of the emulation network includes four layers in general, which are several IoT devices, access points, network routers, and one data-center, orderly arranged from bottom to top in the hierarchy. By default, we have a single data-center connected with 10 routers. Each router is connected with 10 access points. Those access points form a cellular network, serving IoT devices that are randomly scattered in the space. From the perspective of an IoT device, the access latency towards access points, routers and the data-center is set to 5ms, 15ms and 110ms, following a previous work [29].

2) Software Setup: To better estimate the latency, energy consumption and even the start up time of each job, we choose some real-programs for profiling. To simulate `job x`, we adopt the yolo-v3 object detection deployed using darknet framework [31]. Similarly, we choose some off-the-shelf programs to serve as `job a` and `job b`. Specifically, we use a popular OCR tool, namely Tesseract [15] to recognize licence plate numbers. For vehicle attributes recognition, we follow zhao et al's practice [45] and choose SSD algorithm [25] to recognize vehicle attributes, which is deployed using `Caffe` framework.

3) Profiling Methods: We run the prepared programs on a real hardware platforms to collect complete the profiling. To build up the profiling platform, We adopt a desktop PC with i7-6700K CPU, 32GB RAM and GTX-1080ti GPU to serve as the edge server. To simulate the cloud, we use a powerful workstation equipped with Xeon 6148 CPU, 256GB RAM and 4 GTX-2080Ti GPUs. We estimate the energy consumption of each job through multiplying the processing time by the power consumption of the machine. Note that, co-locating more than one jobs in the same edge server may harm the performance, because these jobs tend to compete for resources. We take this into consideration and test the latency of concurrent jobs with various possible combinations.

TABLE II: Latency and energy comparisons.

	Flink 0	Flink 1	Flink 2	**EStream**
Latency `x+a` (t_0)	295 ms	295 ms	295 ms	**295 ms**
Latency `x+a` (t_1)	341 ms	336 ms	449 ms	**305 ms**
Latency `x+b` (t_1)	312 ms	307 ms	276 ms	**276 ms**
Energy (t_0)	47 J	47 J	47 J	**47 J**
Energy (t_1)	85 J	67 J	77 J	**64 J**

C. Benefits of Stream Sharing

The stream sharing mechanism, as introduced in section III-E enables us to reuse streams with negligible overheads. To quantify the benefits, we compare EStream system with Flink like systems by simulating some typical behaviours: at time point t_0, only `job x` and `job a` run in the edge system, the job `x+a` recognizes the plate numbers of all of the detected vehicles. Then at time point t_1, `job b` launches to recognize the vehicle attributes of the detected vehicles, that is to say, we start a new job `x+b`. Note that when using Flink to implement the smart traffic system, there are several ways to deal with the pop-up `Job b`:

Flink 0: Compute-twice: Create a new `job x+b` from scratch. It does not interrupt the existing `job x+a`, but obviously the `x` part is computed twice in two jobs.

Flink 1: Plan-in-advance: There are two jobs in the beginning: `x` and `a`, and they are linked with message queue service. When `job b` comes, it is able to subscribe the result of `job x` instead of computing `job x` twice.

Flink 2: Stop-and-restart: Initially, there is one `job x+a`. When the new `job b` arrives, the system should stop the `job x+a` first, and restart a new `job x+a&b`.

For EStream, we can leverage the stream sharing mechanism, as introduced in section III-E, to reduce both latency and energy consumption when multiple jobs take as input the same stream.

As shown in table II, we evaluate both job latency and energy consumption of processing those jobs. At t_0, all models have the same latency and energy consumption, because only job `x+a` exists in the systems. At time step t_1, `job b` arrives. As we can see, Flink 2 bears the highest latency `x+a`, because of the need to stop `job x+a` and restart a new job `x+a&b`. For Flink 0, since it computes `job x` twice, it poses unnecessary computational burden to edge servers, which not only affects latency `x+a` and `x+b`, but also increase the energy consumption. Flink 1 has the most close performance with EStream. Since it relies on message queue service to share the data of `job x`, there is additional overhead.

D. Decentralized Scheduling

In section IV-F, we have introduced the decentralized scheduling mechanism, which is proposed to solve the scalability problem of distributed edge systems. We compare the algorithm with a centralized one, which traces the status of the computation on each nodes periodically, and tries to provide an optimal scheduling. It suffers from the scale of workloads, which is an intuitive result and thus not presented in the evaluation.

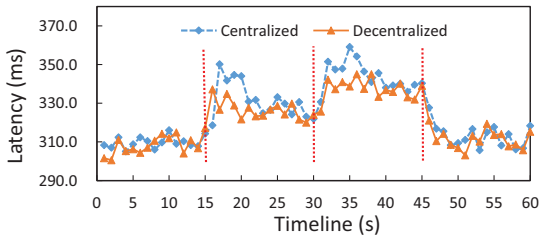


Fig. 12: Decentralized scheduling.

Compared to the basic network topology setup introduced in Subsection V-B, we improve the number of data-centers to four. The computation network consists of 50 routers. The connection among them can be embedded in the plane, forming a planar graph. Other settings remain the same. On average, data packages from the IoT should pass one access points and 2.9 routers before reaching a random data-center. For the centralized method, we choose one of the data-centers to schedule the job. Both of the scheduling methods are evaluated based on the EStream framework.

Initially, each of the data-center runs an `x+a` job, requiring data-sequences from all those IoT devices. Around 15s, two of the data-centers change their job to `x+a&b`. Around 30s, we duplicate the number of IoT devices. Around 45s, the jobs are changed back to `x+a`, and the number of IoT devices is restored to the setup value. Figure 12 shows the average latency of the initial job `x+a`. According to the evaluation, decentralized scheduling performs nearly the same as the centralized one. When the job changes, our method performs a more timely response. When the workload grows abruptly, our method is able to reduce the effect of the burst.

E. Scalability in IoT-Edge-IoT scenarios

EStream performs well in IoT-Edge-Cloud, Cloud-Edge-IoT, IoT-Edge-IoT, and IoT-Edge scenarios. In the IoT-Edge-Cloud and IoT-Edge scenarios, EStream performs computation near the data sources, and thus gains more computation resources and improves the scalability of the system. In the Cloud-Edge-IoT scenario, it works like a content delivery network. The same content is delivered at most N times in a node with N neighbors. These benefits are intuitive and also work for other edge computing systems, so we only mention the IoT-Edge-IoT scenario in this subsection.

In IoT-Edge-IoT applications, both data sources and sinks reside at the edge of the network. We use `job x` with a per-sink grouping for comparison, which is the smart traffic case in Subsection III-D is easy to implement in EStream, but hard for other platforms. For most of other edge computing frameworks, data are collected from data sources to the cloud or a nearby data-center, and the aggregation results are sent to the sinks afterwards. We call this method as All-Cloud. We compare it with EStream in terms of job latency and data traffic that data-center receives. The former metric corresponds to the performance of the system. The latter one reflects the scalability of the system, because of the limitation in the bandwidth and computation capacity of the data-center.

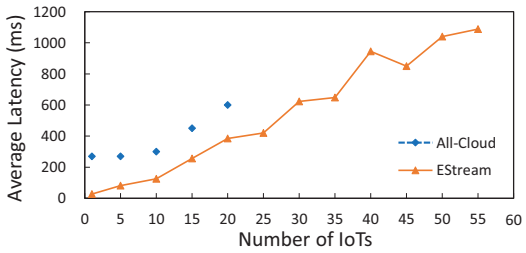


Fig. 13: IoT-Edge-IoT latency.

According to Figure 13, EStream outperforms the All-Cloud method in the aspect of latency. The All-Cloud approach has a static round-trip latency between IoT devices and the cloud, and furthermore, the queuing latency in the cloud increases linearly as the number of IoT grows. In contrast, EStream transmits data from peer to peer, which reduces the overall latency a lot. Figure 14 shows that EStream merely uses the computation resource of the remote cloud. Almost all the transmissions are established from one device towards another, following the nearest routing path. Most of the packages never reach the cloud, so the data traffic around the data-center is reduced to nearly zero.

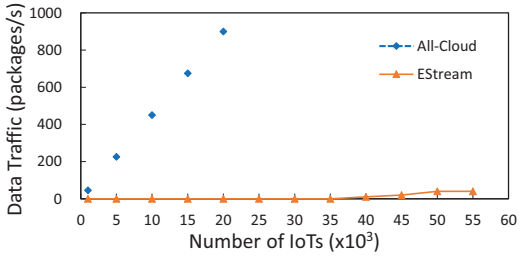


Fig. 14: IoT-Edge-IoT data traffic.

VI. Conclusion

Due to multiple levels of participants and various scenarios, application development in edge computing is more complicated than that in traditional cloud computing. In addition, the new features in these applications further raise more challenges in programming. We try to address these problems using a simple programming model called Edge-Stream. It abstracts data flows as streams to cover different types of scenarios and hide the low level details from programmers. In addition, the data sharing and grouping methods are introduced to support the new features in real deployment. We further discuss implementation details in a prototype design based on Edge-Stream model. It covers job deployment, request propagation, and scheduling, which are critical issues to bring the model into reality. At last, experimental results demonstrate that the framework works efficiently.

Acknowledgment

This work is supported by National Key Research and Development Project of China (Grant No. 2018YFB1003304) and Beijing Academy of Artificial Intelligence (BAAI).

REFERENCES

[1] Bengt Ahlgren, Christian Dannewitz, Claudio Imbrenda, Dirk Kutscher, and Borje Ohlman. A survey of information-centric networking. *IEEE Communications Magazine*, 50(7), 2012.

[2] Alibaba. Alibaba iot service, 2015.

[3] Amazon. Aws greengrass, 2015.

[4] Apache. Apache beam, 2016.

[5] Flavio Bonomi, Rodolfo Milito, Jiang Zhu, and Sateesh Addepalli. Fog computing and its role in the internet of things. In *Proceedings of the first edition of the MCC workshop on Mobile cloud computing*, pages 13–16. ACM, 2012.

[6] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 36(4), 2015.

[7] Craig Chambers, Ashish Raniwala, Frances Perry, Stephen Adams, Robert R Henry, Robert Bradshaw, and Nathan Weizenbaum. Flumejava: easy, efficient data-parallel pipelines. In *ACM Sigplan Notices*, volume 45, pages 363–375. ACM, 2010.

[8] Xu Chen, Lei Jiao, Wenzhong Li, and Xiaoming Fu. Efficient multi-user computation offloading for mobile-edge cloud computing. *IEEE/ACM Transactions on Networking*, (5):2795–2808, 2016.

[9] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[10] Linux Foundation. Edgex project, 2017.

[11] OpenStack Foundation. Starlingx project, 2017.

[12] Nam Ky Giang, Michael Blackstock, Rodger Lea, and Victor CM Leung. Developing iot applications in the fog: a distributed dataflow approach. In *Internet of Things (IOT), 2015 5th International Conference on the*, pages 155–162. IEEE, 2015.

[13] Negin Golrezaei, Karthikeyan Shanmugam, Alexandros G Dimakis, Andreas F Molisch, and Giuseppe Caire. Femtocaching: Wireless video content delivery through distributed caching helpers. In *INFOCOM, 2012 Proceedings IEEE*, pages 1107–1115. IEEE, 2012.

[14] Google. Google cloud iot edge, 2017.

[15] Google. Tesseract-ocr. https://github.com/tesseract-ocr/tesseract, 2020.

[16] Md Munir Hasan, Gobinda Saha, Aminul Hoque, and Md Badruddoja Majumder. Smart traffic control system with application of image processing techniques. In *2014 International Conference on Informatics, Electronics & Vision (ICIEV)*, pages 1–4. IEEE, 2014.

[17] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy H Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *NSDI*, volume 11, pages 22–22, 2011.

[18] Kirak Hong, David Lillethun, Umakishore Ramachandran, Beate Ottenwälder, and Boris Koldehofe. Mobile fog: A programming model for large-scale applications on the internet of things. In *Proceedings of the second ACM SIGCOMM workshop on Mobile cloud computing*, pages 15–20. ACM, 2013.

[19] Ke-Jou Hsu, Ketan Bhardwaj, and Ada Gavrilovska. Couper: DNN model slicing for visual analytics containers at the edge. In Songqing Chen, Ryokichi Onishi, Ganesh Ananthanarayanan, and Qun Li, editors, *Proceedings of the 4th ACM/IEEE Symposium on Edge Computing, SEC 2019, Arlington, Virginia, USA, November 7-9, 2019*, pages 179–194. ACM, 2019.

[20] Yun Chao Hu, Milan Patel, Dario Sabella, Nurit Sprecher, and Valerie Young. Mobile edge computing-a key technology towards 5g. *ETSI white paper*, 11(11):1–16, 2015.

[21] Sabeen Javaid, Ali Sufian, Saima Pervaiz, and Mehak Tanveer. Smart traffic management system using internet of things. In *2018 20th International Conference on Advanced Communication Technology (ICACT)*, pages 393–398. IEEE, 2018.

[22] Jeyhun Karimov, Tilmann Rabl, and Volker Markl. Astream: Ad-hoc shared stream processing. In *Proceedings of the 2019 International Conference on Management of Data*, pages 607–622, 2019.

[23] Jay Kreps, Neha Narkhede, Jun Rao, et al. Kafka: A distributed messaging system for log processing. In *Proceedings of the NetDB*, pages 1–7, 2011.

[24] Juan Liu, Yuyi Mao, Jun Zhang, and Khaled B Letaief. Delay-optimal computation task scheduling for mobile-edge computing systems. In *Information Theory (ISIT), 2016 IEEE International Symposium on*, pages 1451–1455. IEEE, 2016.

[25] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott E. Reed, Cheng-Yang Fu, and Alexander C. Berg. SSD: single shot multibox detector. In Bastian Leibe, Jiri Matas, Nicu Sebe, and Max Welling, editors, *Computer Vision - ECCV 2016 - 14th European Conference, Amsterdam, The Netherlands, October 11-14, 2016, Proceedings, Part I*, volume 9905 of *Lecture Notes in Computer Science*, pages 21–37. Springer, 2016.

[26] Redowan Mahmud, Ramamohanarao Kotagiri, and Rajkumar Buyya. Fog computing: A taxonomy, survey and future directions. In *Internet of Everything*, pages 103–130. Springer, 2018.

[27] Geoffrey Mainland, Matt Welsh, and Greg Morrisett. Flask: A language for data-driven sensor network programs. *Harvard Univ., Cambridge, MA, Tech. Rep. TR-13-06*, 2006.

[28] Microsoft. Azure iot hub, 2015.

[29] Seyed Hossein Mortazavi, Mohammad Salehe, Carolina Simoes Gomes, Caleb Phillips, and Eyal de Lara. Cloudpath: A multi-tier cloud computing framework. In *2nd ACM/IEEE Symposium on Edge Computing (SEC)*, San Jose, CA, October 2017.

[30] Luca Mottola and Gian Pietro Picco. Programming wireless sensor networks: Fundamental concepts and state of the art. *ACM Computing Surveys (CSUR)*, 43(3):19, 2011.

[31] Joseph Redmon. Darknet: Open source neural networks in c. http://pjreddie.com/darknet/, 2013–2016.

[32] Mahadev Satyanarayanan, Paramvir Bahl, Ramón Caceres, and Nigel Davies. The case for vm-based cloudlets in mobile computing. *IEEE pervasive Computing*, 8(4), 2009.

[33] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. Omega: flexible, scalable schedulers for large compute clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 351–364. ACM, 2013.

[34] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. Edge computing: Vision and challenges. *IEEE Internet of Things Journal*, 3(5):637–646, 2016.

[35] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, et al. Storm@ twitter. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 147–156. ACM, 2014.

[36] Athena Vakali and George Pallis. Content delivery networks: Status and trends. *IEEE Internet Computing*, 7(6):68–74, 2003.

[37] András Varga and Rudolf Hornig. An overview of the omnet++ simulation environment. In *Proceedings of the 1st international conference on Simulation tools and techniques for communications, networks and systems & workshops*, page 60. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2008.

[38] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, et al. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing*, page 5. ACM, 2013.

[39] Dale Willis, Arkodeb Dasgupta, and Suman Banerjee. Paradrop: a multi-tenant platform to dynamically install third party services on wireless gateways. In *Proceedings of the 9th ACM workshop on Mobility in the evolving internet architecture*, pages 43–48. ACM, 2014.

[40] Jiarong Xing, Hongjun Dai, and Zhilou Yu. A distributed multi-level model with dynamic replacement for the storage of smart edge computing. *J. Syst. Archit.*, 83:1–11, 2018.

[41] Y. Xiong, Y. Sun, L. Xing, and Y. Huang. Extend cloud to edge with kubeedge. In *2018 IEEE/ACM Symposium on Edge Computing (SEC)*, pages 373–377, Oct 2018.

[42] Ying Xiong, Donghui Zhuo, Sungwook Moon, Michael Xie, Isaac Ackerman, and Quinton Hoole. Amino-a distributed runtime for applications running dynamically across device, edge and cloud. In *2018 IEEE/ACM Symposium on Edge Computing (SEC)*, pages 361–366. IEEE, 2018.

[43] Chao Yao, Xiaoyang Wang, Zijie Zheng, Guangyu Sun, and Lingyang Song. Edgeflow: Open-source multi-layer data flow processing in edge computing for 5g and beyond. *arXiv preprint arXiv:1801.02206*, 2018.

[44] Andrea Zanella, Nicola Bui, Angelo Castellani, Lorenzo Vangelista, and Michele Zorzi. Internet of things for smart cities. *IEEE Internet of Things journal*, 1(1):22–32, 2014.

[45] Y. Zhao, C. Shen, H. Wang, and S. Chen. Structural analysis of attributes for vehicle re-identification and retrieval. *IEEE Transactions on Intelligent Transportation Systems*, 21(2):723–734, 2020.