Received November 13, 2018, accepted December 19, 2018, date of publication December 28, 2018,
date of current version January 16, 2019.

Digital Object Identifier 10.1109/ACCESS.2018.2890005

A Self-Organized Task Distribution Framework for Module-Based Event Stream Processing

SUNYANAN CHOOCHOTKAEW, HIROZUMI YAMAGUCHI, (Member, IEEE), AND TERUO HIGASHINO, (Senior Member, IEEE)
Graduation School of Information Science and Technology, Osaka University, Osaka 565-0871, Japan

Corresponding author: Sunyanan Choochotkaew (sunya-ch@ist.osaka-u.ac.jp)

This work is supported in part by "Japan-US Network Opportunity 2 (JUNO2)," the Commissioned Research of National Institute of Information and Communications Technology (NICT), JAPAN, and JSPS KAKENHI 26220001.

ABSTRACT Tackling bottleneck and privacy issues of cloud computing, we attempt to push event stream processing down to devices which are currently empowered to compute and communicate at the edge of the networks. To accomplish that, we propose a self-organized task distribution framework that is composed of multiple brokers collaborating through our module-based event stream processing engine called *EdgeCEP*. Our system request is event-dependent specified in a brand-new event specification language; still, the event is stored and processed by the relational database. We newly formulate the problem of self-organized task distribution subjective to preferable constraints of computation and communication. The solution for each broker to find individual optimal decision is to apply tabu search with flow-based greedy move regarding pre-ranking flow table. Many experiments are conducted to study and evaluate the performance of the proposed system. The simulation shows that the proposed flow optimization outperforms the naïve algorithm, concretely, 2-times more tasks getting processed and successfully delivered within the same fixed period. The proposed edge-centric method achieves data traffic 7-times less than the cloud-centric approach. The prototype engines have been deployed and evaluated in the real environment.

INDEX TERMS Stream processing, complex event processing, edge computing, self-organized task distribution, Internet of Things (IoT).

I. INTRODUCTION

Toward the world of smart things, superabundant flows of information require processing to discover the hidden meaning behind. Since the 1960s, Cloud Computing comes a long way as a promising solution to handle the exponentially rapid growth of information. Pushing away the heavy computation, such as video stream processing, to the cloud server has become common sense nowadays. Meanwhile, the bottleneck problems due to the limitations of network links and security issues of cloud services significantly cause a high concern.

At the same time, the progress of tiny computation modules is opening the door to break through the limitation of local devices in the past. Such modules now yield a moderate capability to execute complicated processing. User-attaching devices such as smartphones are not only acting as consumers but can also be information providers. On the other hand, the consumers are not limited to those user-attaching devices but could be automatically-actuating modules to control the things like alarm, light, and door. According to [1], 45% of IoT information will be stored, processed, analyzed, and acted at, or close to, the network edge. Due to delay tolerance and security requirements, many applications prefer such kind of environment, for example, security-camera searching for missing child, smart home, smart city, connected health collaborative edge [2], [3]. However, to the best of our knowledge, each of edge computing applications has been developed for only one specific purpose, which is not for general cases. A concrete example is a cooperative video processing in multimedia IoT system from the framework proposed in [4]. With that framework, the whole video will be transferred to the central server even though the camera has enough capability to complete the task itself.

To unlock the capability of drawing computation power down to the edge-device layer, we propose a self-organized task distribution framework for module-based event stream processing on edge called *EdgeCEP*. Through this paper, *edge* refers any computing and network resources along the path from producing sources to consuming destination.

This framework mainly adopts the concept of Complex Event Processing (CEP) that allows devices to be programmable with a serial of module functions, not only a set of logic as found in [5]–[8]. Furthermore, we get rid of the device naming issue as found in the famous module-based tools, like *StreamBase* [9], *Spark* [10], *Storm* [11], and *WSO2* [12], by using event-dependent specifications instead of device-dependent requests. Since the process plan of device-dependent approaches is composed of the source-processor streaming flow graph, a static topology of sources and processors is preferable. In contrast, event-dependent approaches do not require any global registration. An event in the request represents any of those event streams, no matter it comes from which devices. It allows processors of the system to be mobile and dynamic, preferable characteristics of devices at the edge. To achieve that, we design a hybrid module-based processing framework with a new supportive language combining advantages of the conventional-logic specification and the module-processing functionalities.

We consider a system architecture as shown in Fig. 1. Presuming specialists (event composers) contribute on composing event definitions uploaded to the online repositories, subscribers or system users refer to those definitions for generating subscriptions on their system to specify knowledge to be processed as well as responding action. The subscriptions are, then, dispatched to the on-site networks. Over there, multiple *EdgeCEP* devices collaboratively sense, process, and act accordingly to the assigned subscriptions via machine-to-machine (M2M) communication. The processed results may return to the cloud storage for monitoring or even further extending to provide real-time context-aware services as well-designed in [13]. Optimizing cost over such distributed devices should be done differently from general processing distributing systems. Such systems usually assume equity of battery power and energy-consumption balancing is mainly focused [14]. Some edge devices can be stationarily connecting to the power supplier. However, some cannot do that due to the mobility application. In the latter case, energy

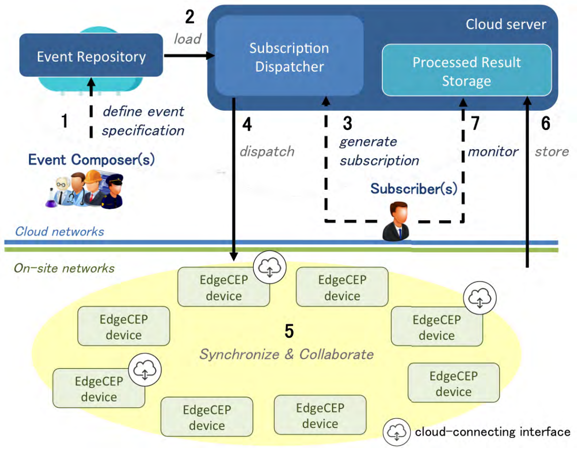


FIGURE 1. EdgeCEP architecture.

contribution becomes a concerning factor. An example of patient anomaly detection and report is illustrated in Fig. 2. In this example, the heart rate (HR) sensor on the patient's smartwatch, unlike an alway-powered camera, cannot perform complicated processing itself. Our idea is to allow data-generator source, HR sensor, to leave processing task to any capable nodes, like a smart bed, along delivering flow to the destination, doctor tablet. Correspondingly, we have newly formulated a flow-based optimizing function over two following assumptions. Firstly, all edge devices can provide preferable constraints of computation and communication contribution regarding its battery limitation. Secondly, devices will be determined as available when it has the potential to finish the task within a specific deadline.

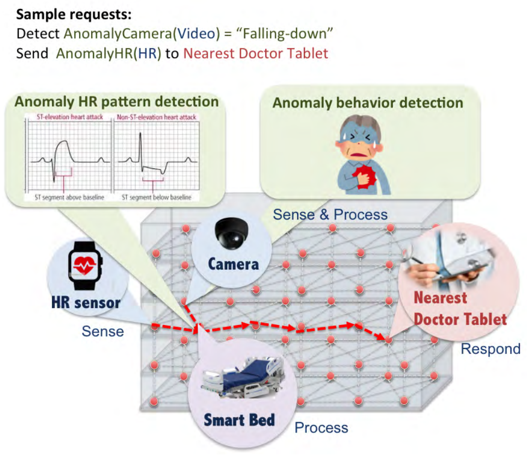

Sample requests:
Detect AnomalyCamera(Video) = "Falling-down"
Send AnomalyHR(HR) to Nearest Doctor Tablet
Anomaly HR pattern detection
Anomaly behavior detection
Sense & Process
Camera
HR sensor
Sense
Nearest Doctor Tablet
Respond
Smart Bed
Process

FIGURE 2. Illustration of framework application.

To observe limitation and present advantages of the proposed framework, we have carried out a scaling simulation. Additionally, to confirm practicality, we have tested our EdgeCEP prototype with real environmental deployment. We installed the prototyped EdgeCEP to the Intel Edison running on top of batman-adv mesh networks [15].

The organization of the paper is as follows. The first section gives the background of Complex Event Processing (CEP). The second section focuses on the task distribution on CEP. Summary of the main contributions is also here. The third section describes the proposed module-based event stream processing engine (EdgeCEP). The fourth section is about the proposed task-distribution framework, problem definition, and solution. The last section reports and concludes the experimental results.

II. COMPLEX EVENT PROCESSING (CEP)

Processing for continuous and timely information has been well-surveyed previously in [16]. In the survey, authors scrutinize 34 related works from 1988 to late 2010 and classify them into three groups: (i) Active database, (ii) Data Stream management, and (iii) Complex Event Processing or CEP. The first group often has scalability issues caused by the growth of rule numbers and the frequency of event arrival rates due to persistent storages and, finally, is almost obsolete now. Meanwhile, the second and third groups are widespread use and investigated until now. Through many great efforts of researchers dedicated so far, the principles defined in [16] are unable to classify those two groups anymore. In other words, the term *CEP* is no more limited to the notification event data models but also covers the stream-based processing tools that support complex event detection as well. In this section, we categorize the related works by the processing-request specification. The first group is CEP with device-dependent specifications. This group requires the name of flow-producing devices, either unique IDs or device contexts such as type and location for making a processing request. The second group is CEP with event-dependent specifications. This group uses a pre-defined event name and attributes instead.

A. CEP WITH DEVICE-DEPENDENT SPECIFICATION

A device-dependent specification commonly applies tuple-based processing approaches. All methods in Data Stream management group classified by [16] fall into this category. All methods in Data Stream management group classified by [16] fall into this category. A producing source flows a stream with pre-defined attributes stored in the relational database. The simplest, mostly generally-used request specification is query representations refer SQL, for example, CQL [17]. There are three fundamental steps to process the continuous flows: (1) windowing (2) processing relation tables, and (3) producing streams from result relations. To window the flow, the source name and available attributes must be pre-defined. Most operations that deal with high-relevant data such as video processing requires comparatively low latency. On top of that, some engines have been introduced specifically for a complicated computational technique such as machine learning [18], [19]. To the best of our knowledge, the existing stream-based software such as *Stream* [17], *Apache Spark Streaming* [10], *Apache Storm* [11], *TIBCO StreamBase* [9], and *WSO2* [12], are deployable only on a centralized node, or clustered nodes with static topology. Unfortunately, most of them have no concern about environmentally-adaptive stream processing to cope with dynamicity of computational/network resource availability due to new query injection, node mobility, and so on.

B. CEP WITH EVENT-DEPENDENT SPECIFICATION

The earliest form of event-dependent specification is a logical statement with *and/or* conjunction of multiple events as found in [5]. Then, the natural event-specification language, e.g. *TESLA*, comes with more flexibility in [6]. All methods in Complex Event Processing group classified by [16] fall into this category. They adopt pattern matching and content filtering along with highly-expressive event specification. Instead of advanced setting as in the stream-based model, producing sources can be discovered by advertisements in runtime. Due to this nature, the event-based model allows fully-distributed deployment. They are usually driven by rules [5], [6], [20] or automata [7], [8]. Formerly, the most classic method is *PADRES* [5], proposed as a rule-based distributed Pub/Subsystems by mapping subscriptions to rules, and publications to facts. We can straightforwardly decompose composite subscriptions if composited events are coming from both sides of a binary tree. Still, in this kind of engines, only conjunction and disjunction operations are available. Next, *RACED* [6] widens expressiveness of the subscription by TESLA language. It makes complex event detection available in a distributed manner, and a master-slave subscription protocol is proposed to reduce the number of non-potential packets. *Adaptive Content-Based Routing in General Overlay Topologies* is, then, proposed in [21] to handle cyclic and dynamic topology. Toward IoT, the efficient rule engine tools aim for large-scale, real-time systems like smart building systems are proposed in [20]. It uses the minimal perfect hash function for filtering and dynamic adaption scheme for blocking non-potential rules. Furthermore, the significance of event derivation certainty is addressed and concerned about designing an efficient and accurate rule-based reasoning mechanism in [22]. However, this kind of engines usually suffers from a spatiotemporal operation, such as aggregation, due to a great cost of computation and memory space compared with the relational-based approach.

C. TASK DISTRIBUTION PROVISION ON CEP

Many proposals attempt to distribute task executions over multiple nodes called brokers to overcome a scalability issue. It is known as multiagent system [23]. In the stream-based architectures, process plans are usually in the form of an operating function graph, considered as *module*. In reference [24], the processing is generalized into two levels of analytics. For the event-based model, most of the processing in distributed systems are assumed to be straightforwardly decomposable (i.e., conjunction, disjunction). Reference [7] states a general framework of task distribution concerning link usage and computational effort under-addressed distribution and detection policies. However, to the best of our knowledge, the existing policies rely on routing protocols and adopt only nearest-to-source heuristic [5], [6], [25]. We consider them as hop-based approaches. Meanwhile, since most of the stream-based engines leave decisions to the users, there are a few research that proposes autonomously distributed mechanisms. One of them is [26], which optimizes operation placement from the given queries. A few state-of-art research have focused on a task mapping and scheduling considering resource limitations in wireless sensor networks. Reference [27] formulates a general algorithm. Some cost functions are regarding energy consumption [14], [28]. Reference [29] proposes a buyer-seller computational task-assignment framework for wireless sensor networks with an auction-based mechanism, similar to the handover algorithm in [30]. Most recently, reference [31] utilizes workload estimation from [32] together with operator-profiling for parallelizing. Reference [33] handles unordered arrival of multiple unsynchronized input sources by newly defining slack-ready tuple to provide a deterministic solution while keeping real-time requirement satisfaction. Reference [34] maps workload partitioning and scheduling in clustered Storm [11] which is stream-based engine into the graph-partitioning problem to come through the high performance of resource utilization which reducing network loads. To support an in-situ expansion of IoT-service requests, [35] proposes an adaptive scale-out mechanism. However, none of the above proposals considers delivery cost from processors to destinations, which is an additional concern for distribution on edge networks. Also, they did not mention the challenges of input sharing and concatenating reference.

D. CONTRIBUTION SUMMARY

This paper is an extension of work earlier presented in [36] with significant enhancement. The primary contributions of this work are summarized as follows.

Firstly, we design a hybrid module-based event stream processing framework aiming for self-organized IoT edge devices over wireless networks. The proposed architecture uses the event-dependent specification like logic-based approaches [5]–[8] but stores and processes with relational databases like general module-based approaches [9]–[12]. It obsoletes the assumption of source knowledge in stream processing while remains the use of relational storage for efficiently processing high-relevant data *in a self-organized manner*. To achieve this, we especially define a new event specification language to express relational operation, and design and develop a broker-based middleware for its distributed execution together with place-and-play API to support customized devices and functions.

Secondly, we newly formulate and solve cost-optimized task assignment and delivery over resource constraints. Unlike previously-proposed task distribution on clusters of fair processors, existing energy-based optimizing function, as found in [14] and [28], may not be appropriate due to the variety of energy contribution capability on distributed-processing units like edge devices. Thus, we define a problem of task assignment and delivery plan to optimize the communication cost under node and link constraints and propose a fully distributed solution. Also, we introduce a dependence-grouping solution to handle an input sharing and concatenating referencing in multiple request environment.

Thirdly, we conduct multiple levels of experiments to study and evaluate the proposed method. The most basic layer is a unit test on the computation module. We test on Intel Edison, a tiny computer module designed for IoT devices. In the aspect of distribution logic, we perform a preliminary logical experiment in the theoretical scenario and a large-scale simulation in the nursing-home use case comparing with centralized and naïve distributing approaches. We also implemented the EdgeCEP prototype and deployed it into Intel Edison for a smart room application enabling the advanced batman protocol [15] for reliable ad-hoc network connection.

III. EdgeCEP: MODULE-BASED EVENT STREAM PROCESSING ENGINE ON EDGE COMPUTING

EdgeCEP is a hybrid event processing engine that is supportive for fully-distributed collaboration of processing on the edge of networks. Referring an architecture shown in Fig. 1, the devices with *EdgeCEP* are considered as *brokers*. There are only three significant conditions to be a broker: (i) computation power (ii) network connection (iii) programmability. Brokers could be a tiny computer module like Intel Edison or a high-efficiency server. Commonly, they are processors. Still, if some sensing modules are attached, they are additionally considered as *sensors*. On the other hand, if they install some actuating modules, they can play *actors* role as well. All brokers are connecting with a self-organized routing protocol like *batman* [15]. Some of them may be Internet gateways. All brokers collaborate to complete all requests, called subscription. The subscription is composed of the ways to detect, analyze, and generate an output as well as an actuator and an action to be activated. It is written in our newly-defined supportive language and committed by *subscribers* (system users) and synchronized with all connecting brokers. It will process the corresponding sensing flows and deliver the outputs to the corresponding actuators to drive a specified action in response.

A. EdgeCEP-SUPPORTIVE LANGUAGE

Inspired by *TESLA* [37] and *CQL* [17], we newly define an expressive event specification language combining syntax for relational operations. A general structure is represented as below:

define	$Name(Att_1, \ldots, Att_n)$ [**aggr**] [**every** T]
$< pre >^*$	
[**case** n:]	
detect	$Pattern(content_1, \ldots, content_m)$
assign	$attr_1 = f_1, \ldots, attr_p = f_p; f = F(content)$
[**consuming**	$e_1, \ldots, e_h; e_i \in content$]
$< post >^*$	
where*	$Att_1 = g_1, \ldots, Att_n = g_n; g = G(attr)$
[***group by***	$(location \mid srcid)$]

*only for aggregation specification (aggr)

To create a specification, there are three steps to follow: (1) address the name and attributes of events at **define** key, (2) state interest pattern of events in terms of a set of contents at **detect** key, and (3) declare how to produce an output from the detected events at **assign** key. Note that, we use the term "content" to denote conjunction of events with values that satisfy the specific conditions. For instance, the content, "Temperature.$val > 40°C$", refers a "Temperature" event with a condition "$val > 40°C$". So, when a sensor detects temperature more than 40°, the generated event will be filtered in this content group. The second and third steps can be done more than one time to define multiple interest patterns in some specifications using the keyword **case**. To illustrate, we assume a pattern of tracking, which corresponds to a *Tracking* sensor deployed at the border between two-sided locations and provides that the direction equals to "0" when someone moves from right to left and "1" when moves from left to right as presented in Fig. 6. Using this, the *MoveIn* specification for an event when someone moves in one location can be defined as below. With a similar specification, we can define the *MoveOut* event in the opposite way.

```
define      MoveIn(location,timestamp)
case 0:
    detect  Tracking(direction = 0)
    assign  location = Tracking.left,
            timestamp = Tracking.timestamp
consuming   Tracking
case 1:
    detect  Tracking(direction = 1)
    assign  location = Tracking.right,
            timestamp = Tracking.timestamp
consuming   Tracking
```

We might use the same contents to produce multiple events. Among those productions, some need to apply them just once to avoid replications. The key **consuming** is applied for specifying which contents will be used just once and then consumed. The keyword time-window, **every** ***T***, is used to designate time to produce in a periodical manner.

For aggregation, there is an aggregation keyword **aggr** to perform assignment and produce an event specified in the **define** clause after the event name and attributes. If the specification is aggregation type, it must specify both $<$ *pre* $>$ and $<$ *post* $>$ parts. In the same way as the map and reduce method, the former part is supposed to be distributed detecting and assigning. Meanwhile, the latter part is for concluding the final results. $<$ *Pre* $>$ part is same as the basic specification mentioned above. $<$ *Post* $>$ part has only one key, **where**, to declare the value-assigning function from $<$ *pre* $>$ to event attributes. The keyword time-window is mandatory to specify a period of $<$ *post* $>$ executions. The following example is the specification of people counts in each location for every 2 minutes. Note that, **group** keyword is used to represent grouping attributes such as *location*, *srcid* at runtime and **outer** stands for the outer join.

```
define      PeopleCount2Mins(location,count,timestamp) aggr
            every 2 mins
< pre >
detect      outer MoveIn and outer MoveOut
assign      location=group.location,
            count=Count(MoveIn)-Count(MoveOut),
            timestamp=Max(MoveOut.timestamp)
< post >
where       location=group.location,     count=Sum(count),
            timestamp=Max(timestamp)
group by location
```

B. BROKER DESIGN

There are three modules included in the EdgeCEP broker: *Content Filterer*, *Task Processor*, and *Coordinator* (see Fig. 3). When an event arrives, the receiver will enqueue them to the matching queue of *Filterer* to filter interest events. Meanwhile, *Coordinator* periodically calculates the distribution plan for task assigning and offloading, and an event notifying using historical records from *Content Filterer*

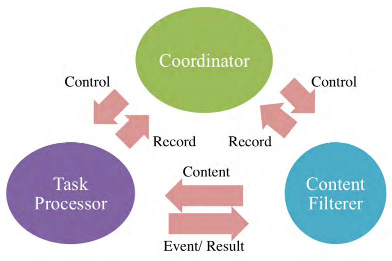


FIGURE 3. Broker component.

and *Task Processor*. Controlled by the calculated distribution plan, *Task Processor* activates only assigned tasks and *Content Filterer* forwards the rest to appointed nodes.

1) TASK PROCESSOR

Task in our work is the processing work that must be accomplished by the forked process named *agent*. A task can be either *definition task* or *subscription task*. We specify the former by the language stated in Section III-A beforehand and available as references. The later is requested by users with a slightly different format, as in the example below. In concrete, the specification of subscription tasks has no attributes and requires a subscriber, *User 1*, with requesting sequence (**seq**), 1, and an actor, *AirController*.

```
Subscription    seq 1 from User1 to AirController
detect          last PeopleCount2Mins(count > 0)
                as PeopleCount and
                AvgTemp5Mins(avg < 20 or avg > 25)
                as AvgTempOut within 5 mins
                from PeopleCount
assign          humid=AvgTempOut.avg,
                location=group.location,
                timestamp=PropleCount.timestamp
consuming       AvgTempOut
group by        location
```

A task forks one agent for each case to allow parallel detecting and assigning operations. For example, according to the task *MoveIn*, two agents will be forked to execute individual cases. One detects *Tracking* event with $direction = 0$ while the other detects an event with $direction = 1$. An agent works as follows. Firstly, it detects a pattern using a state sequence of subscribed contents specified in *detect*. The example shows subscription to send an event when the average temperature over 5 minutes out of comfortable range ($avg < 20$ or $avg > 25$) after detecting some people within the last 5 minutes. Note that, detecting some people is referred by the content *PeopleCount2Mins*($count > 0$) of the previous example event *PeopleCount2Mins*. The sequence is denoted by the keywords **within** and **from**. According to this example, there are two states in sequence, *PeopleCount* and *AvgTempOut*, where *PeopleCount* comes before *AvgTempOut*. Each state owns relational storage to memorize the relevant events. Note that, we use JDBC driver for SQLite connection. Secondly, an agent will periodically check if interesting-pattern sequence reaches final state regarding the time window (T) in *define* specified by the keyword **every**. If it reaches the final state, an agent will generate an event according to the **assign** state. Note that, if time window is not specified, the agent will check final-state reaching for every time that new relevant event inserted. Outputs of definition tasks are pushed back to *Content Filterer* while the outputs of subscription tasks are delivered directly to the destination broker.

2) CONTENT FILTERER

Content Filterer adopts the publish-subscribe mechanism for mapping the general form of an incoming event to *Interest Content* before streaming to corresponding agents in *Task Processor*. For better understandability, we give an example subscription to detect faces of people entering the room written below:

Subscription	**seq** 1 **from** *User*1 **to** *Gateway*
detect	*MoveIn* **and** *Video* **within** 5 mins **from** *MoveIn*
assign	*face*=FaceDetect(*Video*), *location*=group.*location*, *timestamp*=*MoveIn*.*timestamp*
consuming	*Video*
group by	*location*

The cooperation between *Content Filter* and task agents in *Task Processor* is illustrated in Fig. 4. Correspondingly to the example, the main subscription task calls the *MoveIn* task. Agents to process *MoveIn* task will be also activated. One agent subscribes for the event "Tracking (*direction* = 0)" and the other agent subscribes for the event "Tracking(*direction* = 1)" to the content filterer. It derives the identifier, which is composed of source id, location, and event type (any of them might be non-specified), and the corresponding content with checking conditions from the specified pattern. For instance, correspondingly to *FaceDetection* subscription, an identifier *(-,Entrance,MoveIn)* refers any *MoveIn*-type events from any sources in *Entrance* location. An asterisk symbol stands for non-condition. If the identifier already exists in the hash table, the link of that identifier object will append only the new content with checking conditions. If not, a new identifier object will be added to the hash table with the link starting with that corresponding content.

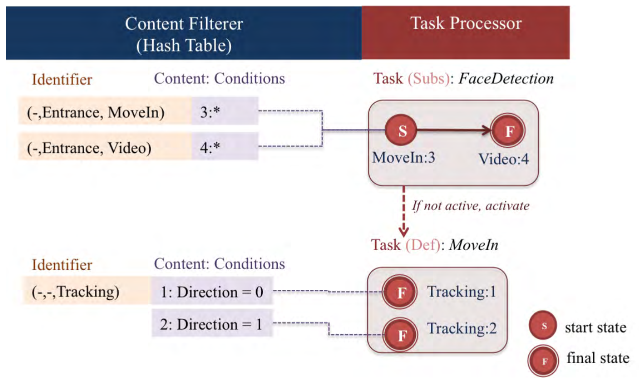


FIGURE 4. Example of content filter and task processor cooperation.

When an event comes, it will extract a set of all possible identifiers that contains at least one composition and compare the hash value. If the identifier object is matching, it will next compare the content condition. If the event content is also matching, it will publish (i.e., enqueue the processing queue) to one or more subscribing agents. In this fashion, the filterer acts as a host with multiple sources. To achieve that, we apply a minimal perfect hash to filter irrelevant events.

3) COORDINATOR

Coordinator performs the optimization algorithm to determine a *distribution plan* that minimize resource utilization while keeping the specified constraints satisfied using statistical record of flows and knowledge about subscription, broker, and networks from *Content Filter*, and statistical record of execution from processing and aggregating agents in *Task Processor*. The distribution plan is applied to control forwarding behavior of *Content Filter* and processing behavior of *Task Processor*. The assigning part in task-distribution plans can affect the neighbor decision on local optimization. It sends this assignment part to the neighbors as information as well. The communication flow is depicted in Fig. 5. Our approach is not only based on distance heuristics, like nearest to the source or shortest path to destinations but also considers resource constraints and input flow volume (arrival rate). Fig. 6 shows an example of a simple distribution plan to cut face images from video stream when someone moved into the interest location. In other words, the collaborating task is to process the subscription given in the above subsection (*III-B.2 Content Filter*). Regardless of all constraints, the tasks, *MoveIn* and *Face detection*, are both assigned by the above heuristics. The particular problem definition and the solution are described in Section IV.

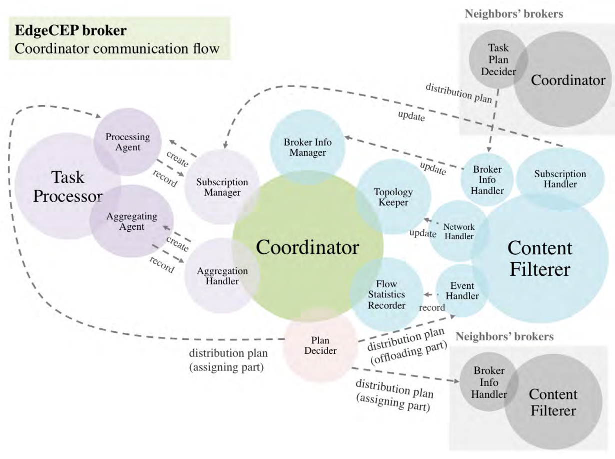


FIGURE 5. Coordinator communication flow.

C. EdgeCEP API

To allow any devices connected to the EdgeCEP engines, we develop a Java API, *Event Driver*, for users to interpret the sensing streams from their devices to the event entity.

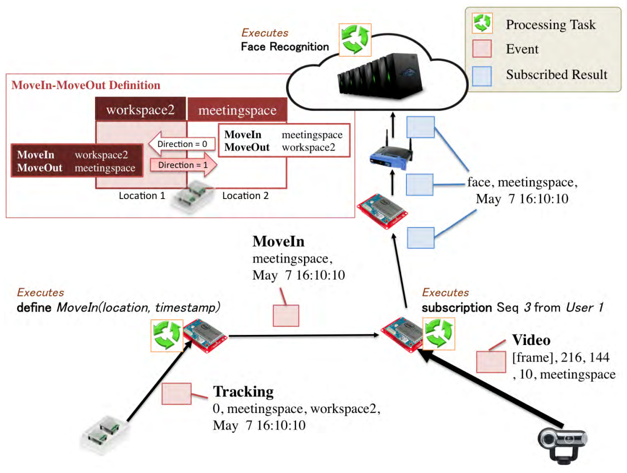
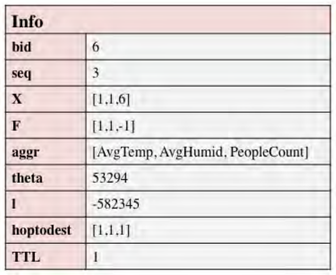


FIGURE 6. Task distribution: face-recognition example.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<definition type="composite" name="MoveIn" desc="move-in detection">
    <attribute name="srcid" type="TEXT"/>
    <attribute name="location" type="TEXT"/>
    <attribute name="timestamp" type="TIME"/>
    <case>
        <assign attribute="location" function="assign">
            <input id="Tracking" attribute="location"/>
        </assign>
        <assign attribute="timestamp" function="assign">
            <input id="Tracking" attribute="timestamp"/>
        </assign>
        <unitpattern id="Tracking" definition="Tracking">
             <store type="now"/>
            <condition>direction == 0</condition>
        </unitpattern>
        <consume>Tracking</consume>
    </case>
    <case>
        <assign attribute="location" function="assign">
            <input id="Tracking" attribute="location2"/>
        </assign>
        <assign attribute="timestamp" function="assign">
            <input id="Tracking" attribute="timestamp"/>
        </assign>
        <unitpattern id="Tracking" definition="Tracking">
             <store type="now"/>
            <condition>direction == 1</condition>
        </unitpattern>
            <consume>Tracking</consume>
     </case>
</definition>
```

FIGURE 7. Example of MoveIn event definition stored at broker.



Info	
bid	6
seq	3
X	[1,1,6]
F	[1,1,-1]
aggr	[AvgTemp, AvgHumid, PeopleCount]
theta	53294
I	-582345
hoptodest	[1,1,1]
TTL	1

FIGURE 8. The 3rd broadcast *Info* message from node 6.

It can refer to the existing definition or form a new one. The connecting brokers must update the new implemented classes to execute. They have to suspend the processing and re-build the EdgeCEP program.

1) EVENT DRIVER

The driver class for connecting sensors to the EdgeCEP is an extension of the abstract class, *Driver*, by implementing the following functions.

Driver(String *driverName*, String *eventType*)
boolean **init**()
Object[] **getValues**()

The constructor must specify the name of the driver to be referred and the producing event type. An initial function, *init*, is called at starting state. A *getValues* will repeatedly run until interrupted in the separate thread.

2) PROCESSING FUNCTION

Beyond standard functions, any Java application is eligible to be included in the EdgeCEP as an extension by just extending a class called *Function*. There are two abstract-functions to implement:

Object **execute**(List *types*, Object[] *values*)
Type **getDefaultType**()

An *execute* function will be activated when producing the events by inputting an array of memorized values and their corresponding type. A *getDefaultType* function is to refer the output type.

IV. SELF-ORGANIZED TASK DISTRIBUTION FRAMEWORK

The proposed framework is composed of multiple broker nodes that are collaborating on processing via self-configured and self-organized ad-hoc networks. The global knowledge of user, i.e. subscriptions and event definitions stored in *XML* format (see example in Fig. 7), are supposed to be synchronized. Each broker will periodically update its decision and broadcast the *Info* message for every specific window time. The info message contains the information that is necessary for others to calculate the distribution plan, as will be addressed in the problem definition below. We give an example of info message in Fig. 8. In this section, we formulate a task distribution problem and introduce a sub-optimal searching algorithm to solve in general as well as procedures to deal with aggregation and task-dependency issues.

A. *TASK DISTRIBUTION PROBLEM DEFINITION*

As defined above, *task* represents composite definition and subscription. Subscription tasks are always active while definition tasks will be active only when any subscription tasks refer to them. A distribution plan is composed of (i) the assigning matrix, designating which processing task is going to be active on which broker node, (ii) the offloading matrix, showing paths from the stream-source broker to the corresponding assigned broker, and (iii) the notifying matrix, showing paths from the processor to the destination actuator. In this part, we formally give a problem definition to minimize resource utilization on edge networks concerning the total flow volume from sources to destinations of all active tasks under resource and consistency constraints.

Problem 1 (Global Task Distribution): Consider the edge network of a set of bi-directional connecting broker nodes, B. Presume global knowledge: (1) distance matrix $H = (h_{qps}) \in \mathbb{Z}^{|B|\times|B|\times|B|}$ as hop count from node q to node s via node p, (2) link-constraint matrix $L = (l_q) \in \mathbb{Z}^{|B|}$ as link capacity of node q, and (3) node constraint $\Theta = (\theta_q) \in \mathbb{Z}^{|B|}$ as processing capability of node q.

Given a set of processing tasks, T, with a specified destination node for each task, denoted by $D(\mathrm{T})$, and a generated content set, C. Let matrix $A = (a_{ij}) \in \{0,1\}^{|\mathrm{T}|\times|C|}$ denote mapping of task i ($\tau_i \in \mathrm{T}$) to content j ($c_j \in C$). Suppose statistical predictions for each node q to provide (1) $\Phi = (\phi_{qj}) \in \mathbb{Z}^{|B|\times|C|}$ as the input flow volume of c_j, (2) $\Psi = (\psi_{iq}) \in \mathbb{Z}^{|\mathrm{T}|\times|B|}$ as the output flow volume of τ_i and (3) $\Omega = (\omega_{iq}) \in \mathbb{Z}^{|\mathrm{T}|\times|B|}$ as the execution cost of τ_i. We define assigning matrix $X_{\mathrm{T}\times B\times B}$, corresponding offloading matrix $Y_{\mathrm{T}\times B\times B\times B}$, and notifying matrix $Z_{\mathrm{T}\times B\times B\times B}$ as follows.

Presume a routing protocol to provide a set of nodes in an momentary optimized path from any node q to any node t when applying assigning matrix X, denoted by $Path(X, q, t)$.

$$X_{\mathrm{T}\times B\times B} : x_{iqt} = \begin{cases} 1; & \textit{if task } i \textit{ of node } q \textit{ is entrusted to node } t \\ 0; & \textit{otherwise} \end{cases}$$

$$Y_{\mathrm{T}\times B\times B\times B} : y_{iqst} = \begin{cases} 1; & \textit{if } x_{iqt} = 1 \textit{ and node } s \in Path(X, q, t) \\ 0; & \textit{otherwise} \end{cases}$$

$$Z_{\mathrm{T}\times B\times B\times B} : z_{iqst} = \begin{cases} 1; & \textit{if } x_{iqt} = 1 \textit{ and node } s \in Path(X, t, D(\tau_i)) \\ 0; & \textit{otherwise} \end{cases}$$

Define a coefficient matrix $E = (e_{sqtj}) \in \{0,1\}^{|B|\times|B|\times|B|\times|C|}$ to specify whether the node s is in the path to deliver content c_j from source node q to assigned node t or not, formulated as:

$$E : e_{qstj} = \begin{cases} 1; & \sum_{i=1}^{|\mathrm{T}|} y_{iqst} \cdot a_{ij} \geq 1 \\ 0; & otherwise \end{cases}$$

Therefore, the problem is to find X with the following objective function and constraints.

$$Minimize \sum_{q\in B}\sum_{s\in B}\sum_{t\in B}\left\{\sum_{j=1}^{|C|} e_{qstj}\phi_{qj} + \sum_{i=1}^{|\mathrm{T}|} z_{iqst}\psi_{iq}\right\}$$

subject to

(1) resource constraints:

$$\forall_{q\in B} \sum_{i=1}^{|\mathrm{T}|} x_{iqq} \cdot \omega_{iq} \leq \theta_q$$

$$\forall_{s\in B} \sum_{q\in B}\sum_{t\in B}\{\sum_{j=1}^{|C|} e_{qstj}\phi_{qj} + \sum_{i=1}^{|\mathrm{T}|} z_{iqst}\psi_{iq}\} < l_s$$

(2) consistency constraints:

$$\forall_{i\in\mathrm{T}}\forall_{q,s\in B;q\neq s}\, x_{iqs} \leq x_{iss}$$

To solve the problem above, all brokers need to obtain the up-to-date global knowledge about topology and link/node capacities. Since such information is often unachievable in practical, we additionally define another problem for individual nodes to solve with limited local knowledge as below.

Problem 2 (Local Task Distribution): Given that each node q with processing capacity θ'_q maintains a set of local members $G = \{q\} \cup \{s; h_{qqs} \leq r\}$, where r is a vicinity-coverage degree. Presume a routing protocol to provide (1) a set of nodes in a momentary path from host node to any local member $t \in G$ when applying assigning matrix X, denoted by $Path'(X, t)$, and (2) the number of hops to reach the destination of each task via each local member, denoted by $hoptodest_G^{\mathrm{T}}$. We define the local distribution plan (X', Y', Z') as follows:

$$X'_{\mathrm{T}\times G} : x'_{it} = \begin{cases} 1; & \textit{if task } i \textit{ is entrusted to node } t \\ 0; & \textit{otherwise} \end{cases}$$

$$Y'_{\mathrm{T}\times R_g\times G} : y'_{ist} = \begin{cases} 1; & \textit{if } x'_{it} = 1 \textit{ and node } s \in Path'(X, t) \\ 0; & \textit{otherwise} \end{cases}$$

$$Z'_{\mathrm{T}\times G} : z'_{it} = \begin{cases} hoptodest_t^{\tau_i}; & \textit{if } x'_{it} = 1 \\ 0; & \textit{otherwise} \end{cases}$$

Correspondingly, we define the local coefficient matrix $E' = (e'_{stj}) \in 0, 1^{|G|\times|G|\times|C|}$. $e'_{stj} = 1$ if local node s is in the path to offloading-target node t. Otherwise, $e'_{stj} = 0$. Provide local knowledge at stable state of $s \in G - \{q\}$: (1) residual link capacity l'_s, (2) residual processing capability θ'_s (3) assigning plan $X'^s = (x^s_{it}) \in \{0,1\}^{|\mathrm{T}|\times|G^s|}$, when G^s denotes the local-member set of node s.

Similarly to Problem 1, we assume statistical method to provide (1) $\Phi' = (\phi'_j) \in \mathbb{Z}^{|C|}$ as the momentary input flow volume of c_j, (2) $\Psi' = (\psi'_i) \in \mathbb{Z}^{|\mathrm{T}|}$ as the expected output flow volume of τ_i, and (3) $\Omega' = (\omega'_i) \in \mathbb{Z}^{|\mathrm{T}|}$ as the average execution cost of τ_i. The momentary input flow is accumulation of flows originated at the current node and flows forwarded by the other nodes. Since a path from assigned node to the task destination is not always determined, the maximum link-capacity increment at any node $s \in G - \{q\}$ is a summation of known-path input flows ($\sum_{t\in G}\sum_{j=1}^{|C|} e'_{stj}\phi'_j$) and unknown-path output flows ($\sum_{t\in G}\sum_{i=1}^{|\mathrm{T}|}(1-x'_{is})\psi'_i$). For momentary consistency, if we assign the task i to node $t \in G$, the node t must already decide to execute task i ($x^t_{it} = 1$).

Therefore, the problem is to find the above-defined X' that

$$Minimize \sum_{t\in G}\left\{\sum_{s\in G}\sum_{j=1}^{|C|} e'_{stj}\phi'_j + \sum_{i=1}^{|\mathrm{T}|} z'_{it}\psi'_i\right\}$$

subject to

(1) node constraints: $\forall_{t\in G} \sum_{i=1}^{|\mathrm{T}|} x'_{it} \cdot \omega'_i \leq \theta'_t$

(2) link constraints: $\forall_{s\in G-\{q\}} \sum_{t\in G}\left\{\sum_{j=1}^{|C|} e'_{stj}\phi'_j + \sum_{i=1}^{|\mathrm{T}|}(1 - x'_{is})\psi'_i\right\} \leq l'_s$

(3) momentary consistency constraints: $\forall_{t\in G-\{q\}}\forall_{i\in\mathrm{T}}$ $max(1 - x'_{it}, x^t_{it}) = 1$

B. TASK DISTRIBUTION SOLUTION

The task distribution problem can be reduced to the Generalized Assignment (MINGAP) problem by fixing all tasks with a single input from only one source with multiple constraints. According to [38], MINGAP is equivalent to a Generalized Assignment Problem (GAP). The partition problem of a given set of positive integers into two equal-sized subsets can be reduced to GAP when assigning weight-fixed items to two just-enough equal-capacity knapsacks. Correspondingly, the above-defined problem is NP-hard. Since optimal solutions usually consume too much time to retrieve, we propose a method to find a sub-optimal solution from local knowledge regarding Problem 2 enhanced with pre-estimation ranking working as the following explanation.

Every node s periodically broadcasts its status, containing constraints (l'_s and θ'_s) and assigning matrix (X'_s). For every specific period of time or when interrupted by task update, each node q executes the tabu-based searching algorithm with knowledge of tasks ($\mathrm{T} = (C, A, D)$), candidate nodes (G), statistics ($stat = (\Omega', \Phi', \Psi')$), constraints ($constraint = (l'_{s\in G-\{q\}}, \theta_{q\in G}, X'_{s\in G-\{q\}})$), and routing protocol ($RoutingProtocol = (Path', hoptodest)$), as simply written in Algorithm 1, to update its new assigning matrix. Tabu search allows users to limit the searching round to avoid too-much resource consumption. We additionally apply a greedy selection algorithm for a better sub-optimal solution in such round-limit search. Because the $Path$ function requires X', the exact flow volume cannot be computed before knowing that. However, we compute an estimated flow volume, $W_{\mathrm{T}\times G} : w_{it}$, when assign a task τ_i to a specific node $t \in G$ by using the basic assigning matrix (X^{base}), which is to assign all tasks to the host node, with the following function.

$$w_{it} = (\sum_{j=1}^{|C|} a_{ij}\phi'_j)|Path'(X^{base}, t)| + \psi'_i hoptodest_t^{\tau_i}$$

For each searching-round, it will find the change of the task that can validly get minimum flow volumes applied for the next search. For each task i, it will try changing the assignment to node t in ascending order of the estimated volume ($W_{\mathrm{T}\times G}$). If the changed X' (i.e., X^{tmp} in the pseudo code) is in the tabu list, it will continue trying the next order. If not, it will compute the validity and volume by the formulation in Problem 2 and add that change to the tabu list.

To avoid loop-forwarding, at node q, the task i cannot be offloaded to node t if it meets all of the following conditions in the same time: (1) node t currently offloads task i to itself (2) previous X used to offload task i to node t. If there is no feasible solution in the iteration, we adapt the algorithm to choose X' that does not violate loop-forwarding condition and has the minimal total violation cost ($\sum_{t\in G} Violate_t^{total}$), where

$$Violate_t^{total} = \frac{Violate_t^{\Omega}}{\theta'_t} + \frac{Violate_t^{\Phi}}{l'_t};$$

Algorithm 1 Pseudo Code of Local optimization

```
For an arbitrary node $q$:
Input: $\mathrm{T}, G, stat, constraint, RoutingProtocol$
Output: $X$
$X^{base}_{\mathrm{T}\times G} : x^{base}_{it} = \{1|t = q, 0|otherwise\}$;
$W_{\mathrm{T}\times G} : w_{it} =$
$(\sum_{j=1}^{|C|} a_{ij}\phi'_j)|Path'(X^{base}, t)| + \psi'_i hoptodest_t^{\tau_i}$;
$min^{global} \leftarrow \infty$;
$tabu \leftarrow \{\}$;
$X^{cur} \leftarrow X^{base}$;
$X^{min} \leftarrow X^{base}$;
while $X^{cur} \neq$ null and !stopCondition() do
    $min^{local} \leftarrow \infty$;
    for $i = 1$ to $|\mathrm{T}|$ do
        $X^{tmp} \leftarrow$ copy $X^{cur}$;
        $queue \leftarrow$ enqueue $W[i, ]$;
        while $queue \neq \phi$ do
            sort $queue$ ascendingly;
            $first \leftarrow$ dequeue $queue$;
            if $X^{tmp}_{it} = 0$ then
                set $X^{tmp}_{it} = 1$ and $X^{tmp}_{is|s\neq t} = 0$;
                if $X^{tmp} \notin tabu$ then
                    add $X^{tmp}$ to $tabu$;
                    $valid \leftarrow$ validate $X^{tmp}$ with
                    $constraint$;
                    $volume \leftarrow$ find objective value of
                    $X^{tmp}$;
                    if $valid \wedge (volume < min^{local})$ then
                        $X^{next} \leftarrow X^{tmp}$;
                        $min^{local} \leftarrow volume$;
                    end
                    break;
                end
            end
        end
    end
    $X^{cur} \leftarrow X^{next}$;
    if $min^{local} < min^{global}$ then
        $X^{min} \leftarrow X^{cur}$;
        $min^{global} \leftarrow min^{local}$;
    end
end
$X = X^{min}$;
```

In the aspect of computation complexity, we use n, k, y and p that denote the number of tasks, the number of nodes, the bounded number of iterations and possibilities to move for each iteration respectively. The complexity of the pre-calculating program is $O(npk)$. The complexity of the greedy move depends on volume estimation is $O(pk)$. Accordingly, the complexity of our algorithm is $O(y(npk+pk)) = O(ynpk)$ for each of individual nodes. Meanwhile, the complexity of centralized full-searching is $O((pk)*k!)$.

C. AGGREGATION HANDLING PROCEDURE

For aggregation tasks, we cannot determine the results until knowing all relevant data collected from all brokers. In this paper, we use the term, *aggregation task*, to stand for such a kind of tasks. In common, the aggregation task needs to perform on a specific node. However, in distributed systems, the aggregation is usually divided into two steps. The first step is to distributedly execute while the second step is to finish at one node. In the similar way, we allow distribution by *pre-post* specification, mentioned in Section III-A. In the similar way, we allow distribution by *pre-post* specification, mentioned in Section III-A. To decide the header node for the second step, we exploit a blind bid protocol that leads to only one agreement by evaluating the values of all candidates from topology and identification. We suppose to produce the aggregation result periodically. The header node will make a request and wait for all replies with a specific timeout to execute post-processing.

D. TASK-DEPENDENCY HANDLING PROCEDURE

There are some cases that more than one subscription require processing on the same event streams. For clearer understanding, we give an example of the system running the three following subscriptions: *Stranger Notification* to notify when unrecognized persons enter the house, *Members' Entrance Record* to keep records when recognized persons, i.e.g, a member of the house, come back and *Anomaly* to detect anomaly behavior in the house. We can construct a directed acyclic graph (DAG) to show the dependency of activating tasks, including subscription tasks and referred to definition tasks as shown in Fig. 9. Roots are atomic events while leaves are subscription tasks. Edges start from referred events (parents) or definition tasks to referring to tasks (children). Since the *Stranger Notification* and *Members' Entrance Record* subscription tasks require (or depend on) the *Face Label* task, they are both children of *Face Label*. In other words, they are sharing the dependency.

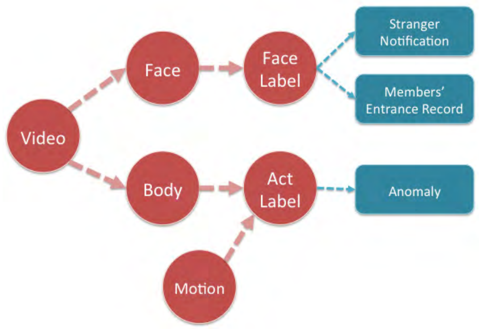


FIGURE 9. **Example of task dependency.**

Assigning tasks without considering the dependency might cause these two following problems: (1) early consumption and (2) ambiguous passing. For the former, one task might early consume a required event of other tasks. For example, when a node with the *Face Label* stream decides to execute *Stranger Notification* task but assign *Members' Entrance Record* task to a neighbor node. Since the *Face Label* stream will be consumed for the *Stranger Notification* task, *Members' Entrance Record* task will never be inputted. The latter is how to forward the event that is subscribed by two or more tasks assigned to different nodes.

Without replication, which coming with extra resource consumption, we introduce a dependency-grouping technique ensuring the dependency-sharing tasks to execute on the same node. Applying the task distribution solution in subsection IV-B, we consider an assigning and passing sets of tasks instead of individual task i. Starting with fixed task members assigned to itself, it constructs an assigning set. It further includes the dependency-sharing tasks of those fixed members. The dependence-sharing tasks include (1) parents, (2) parents' one-hop children (siblings), and (3) dependence-sharing tasks of parents and siblings. In the example given in Fig. 10 (left), with fixing a *Face Label* task, *Face* (parent) is included in the same dependency-aware set. Also, *Video* and *Body*, which are *Face*'s parent and sibling, are included too. An example for two fixed members of assigning set is presented in Fig. 10 (right). The assigning restriction deals with the early-consumption issue.

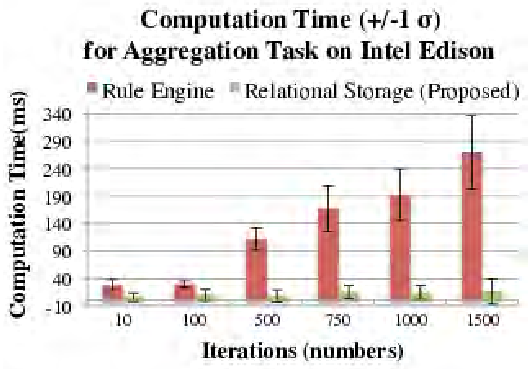


FIGURE 10. **Example results from the dependency-grouping technique (Right: an assigning set by fixing {*Face Label*}, Left: an assigning set by fixing {*Face Label, Act Label*}).**

To avoid the ambiguous passing issue, we group the tasks that are not in the assigning set into multiple sets for further passing. Tasks will be in the same passing set if they are relatives regardless of hops inside the assigning set, named descendants. Accordingly, if the task i_1 offloaded to node $t_1 \in G$ and i_2 offloaded to node $t_2 \in G$ are in the same passing set, they must be passed to the same node ($\forall_{s \in R_g}\ y'_{i_1 s t_1} = y'_{i_2 s t_2}$).

V. IMPLEMENTATION AND EXPERIMENTAL RESULTS

We evaluate our proposed system in four perspectives: (i) unit design comparing the performance of processing between the conventional rule engine and proposed relation-based engine, (ii) task distribution and delivery algorithm in the preliminary experiment, (iii) scalability of distribution solution by large-scale simulation in the theoretical scenario and practical nursing home scenario, and (iv) utility of prototype program in the

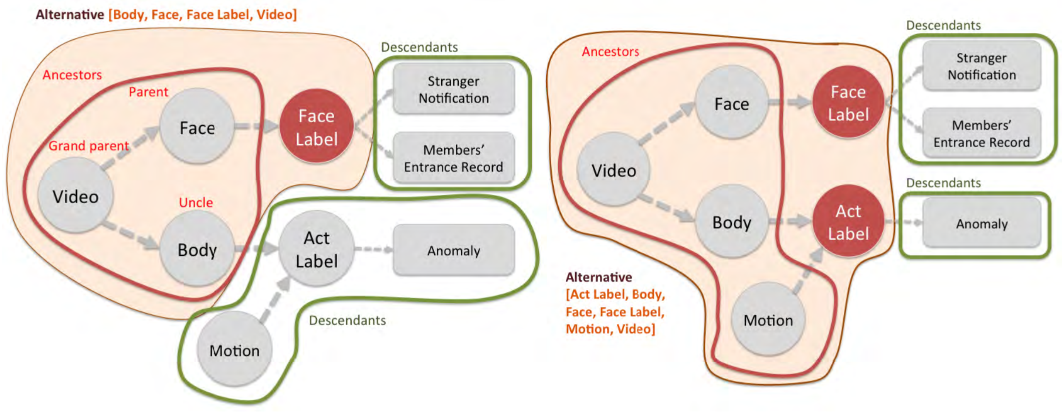


FIGURE 11. Computation time of aggregation task on Intel Edison.

real environment. Through all experiments, we fixed the stop condition for searching at 20 rounds.

A. UNIT TEST

To affirm our broker design, we develop two simple engines, one with our proposed relation-based processing component and the other with the JESS rule-engine [39]. Both engines are installed on the same Intel Edison board and run a simple aggregation task finding an average of temperature data within last x seconds defined as following:

$$\begin{aligned} &define \quad MonitorAvgTemp(avgTemp : FLOAT) \\ &from \quad Temp()\ range\ x\ secs \\ &where \quad avgTemp = avg(Temp.val) \end{aligned}$$

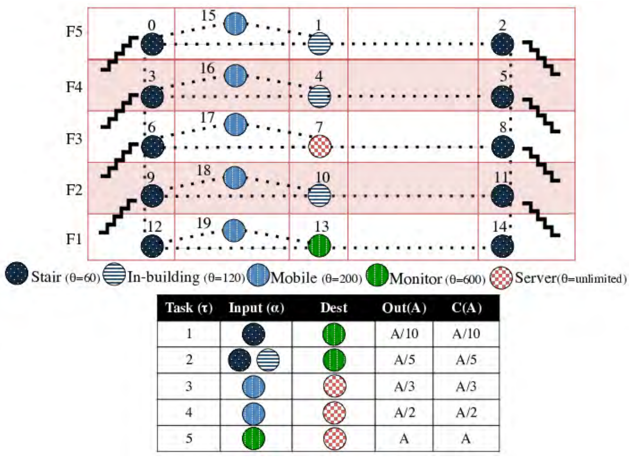
As expected, with a variation of iteration numbers from different range x, the result in Fig. 11 shows that a pure rule-based approach needs extremely-expensive computation cost for a high number of iterations. Meanwhile, the cost of the combination approach remains low even if the number of iterations is high. At 1500 iteration numbers, an average computation time of rule-engine approach is 16.9 times higher than the relation-based approach.

B. PRELIMINARY EXPERIMENT

To observe the radius factor of local knowledge, we implement JAVA program of the proposed flow-based algorithm and test under the controlled scenario as presented in Fig. 12. We omit the link constraints in this experiment to reduce the complexity of the communication path.

With different producing rates of the source nodes, including *Stair*, *In-building*, *Mobile and Monitor*, total flow volumes per time slot are summarized in Table 1. The results confirm that available knowledge, reflected by the radius of the vicinity group, affect the distribution decisions as well as the total flow volume.

In particular, there are three interesting findings. Firstly, insufficient knowledge not only leads to an extremely-large flow volume but can also cause an infinite loop due to the



Task (τ)	Input (α)	Dest	Out(A)	C(A)
1	●	🟢	A/10	A/10
2	● ▤	🟢	A/5	A/5
3	🔵	▦	A/3	A/3
4	🔵	▦	A/2	A/2
5	🟢	▦	A	A

FIGURE 12. Logical test scenario.

TABLE 1. Summary table of total flow volume for various radius at each producing-rate scenarios.

	Radius					
Producing rate	1	2	3	4	5	6
100	**73,440**	41,280	41,280	41,280	41,280	41,280
150	27,623	27,515	26,555	26,555	26,555	26,555
200	16,670	16,670	**16,950**	16,670	16,670	16,670
250	10,824	10,824	10,824	10,824	10,824	10,824
300	5,650	5,650	5,650	5,650	5,650	5,650

firstly-step wrong decision. For instance, as seen Fig. 13, where *producing rate* = 100, some tasks are forwarded in a cycle path when only one-hop knowledge is available (*r=1*). As a result, the number of total flows continuously increases over time. On the other hand, when expanding the knowledge to two-hops, the whole processing can reach the best stable state (i.e., when the total flow volume equals to the total flow volume from full knowledge without fluctuation). Secondly, the better solution needs a larger total volume at early steps because it assigns the node farther from source for the better result in the overall path. According to Fig. 14 when

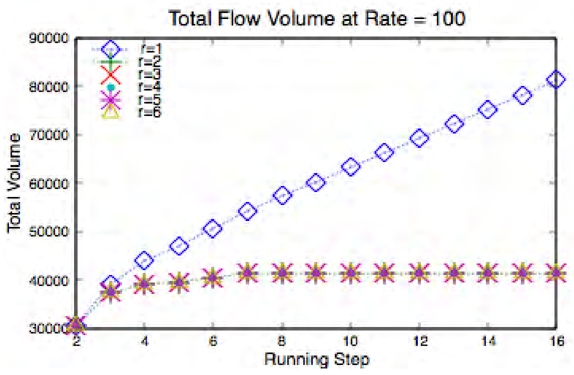


FIGURE 13. Logical test: particular result when producing rate = 100.

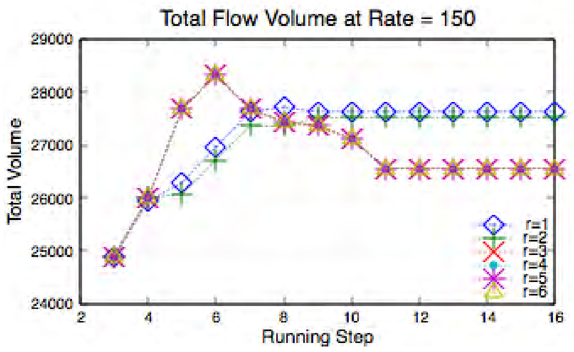


FIGURE 14. Logical test: particular result when producing rate = 150.

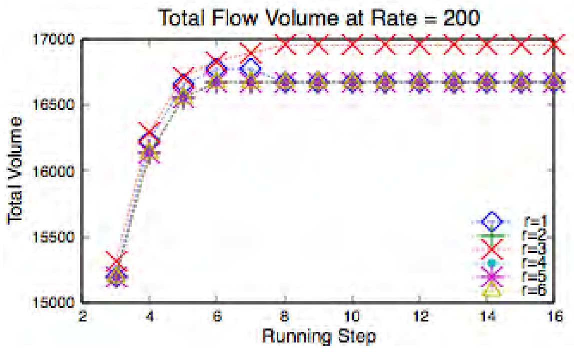


FIGURE 15. Logical test: particular result when producing rate = 200.

producing rate = 150, the best solutions gain larger flow till running step 8. Thirdly, the larger radius is not always a better decision. According to the result in Fig. 15 when *producing rate* = 200, *radius* = 3 obtains inferior solution due to incomplete knowledge while the smaller-radius runs achieve better solution because of fortunate random-pushing.

As shown in the trend line (Fig. 16), the higher radius causes the larger overhead packets with slow increment. Thus, the possibility to achieve the best solution must be traded off with the overhead downside. For example, in the test scenarios, beyond the radius that covers the resource-abundant server (*radius* = 4), the best solution could be guaranteed.

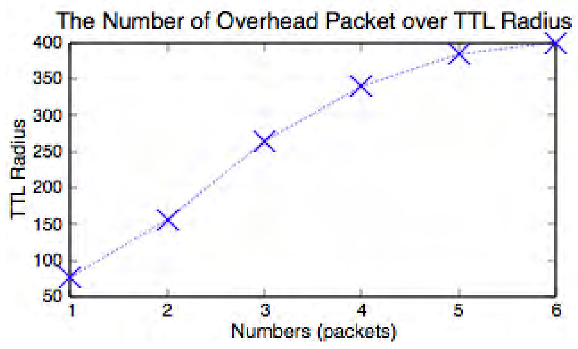


FIGURE 16. Overhead trends over various radius.

C. SIMULATION

1) THEORETICAL SCENARIO

In the theoretical scenario, we assume six types of nodes to produce the data flow as shown in Table 2, where $\Omega(T_x)$ is the benchmark time to complete task T_x for one input. The fixed and blank nodes will be arranged in square grid topology while the mobile nodes will be moving around all covering areas by Random-waypoint mobility model with the random speed from 2 to 20 m/s as shown in Fig 17.

TABLE 2. Node types in theoretical scenario.

Type	Data Size (KB)	Sampling T (s)	Capability (μs)
fixed S	10	1	$\Omega(T_1) \times 100\%+\epsilon$
fixed M	10	0.5	$2 \times \Omega(T_2) \times 50\%+\epsilon$
fixed L	500	1	$\Omega(T_3) \times 100\%+\epsilon$
mobile M	10	0.5	$2 \times \Omega(T_4) \times 50\%+\epsilon$
mobile L	500	1	$\Omega(T_5) \times 50\%+\epsilon$
(blank)	0	0	$2 \times \Omega(T_2) \times 100\%+\epsilon$

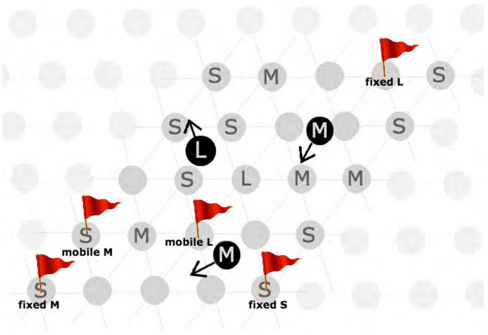


FIGURE 17. Theoretical scenario.

The benchmark (Ω) is computed as shown in the equation (1). In this simulation, we set *CPI* and *Clock Rate* as 2.5 and 500 MHz, respectively.

$$\Omega(T_x) = \frac{CPI \times Intruction\ Count_{T_x}}{Clock\ Rate} \tag{1}$$

We assume five tasks to process the input from each fixed and static nodes. The reduced ratio and instruction count depend on the data size that they produce as shown in Table 3.

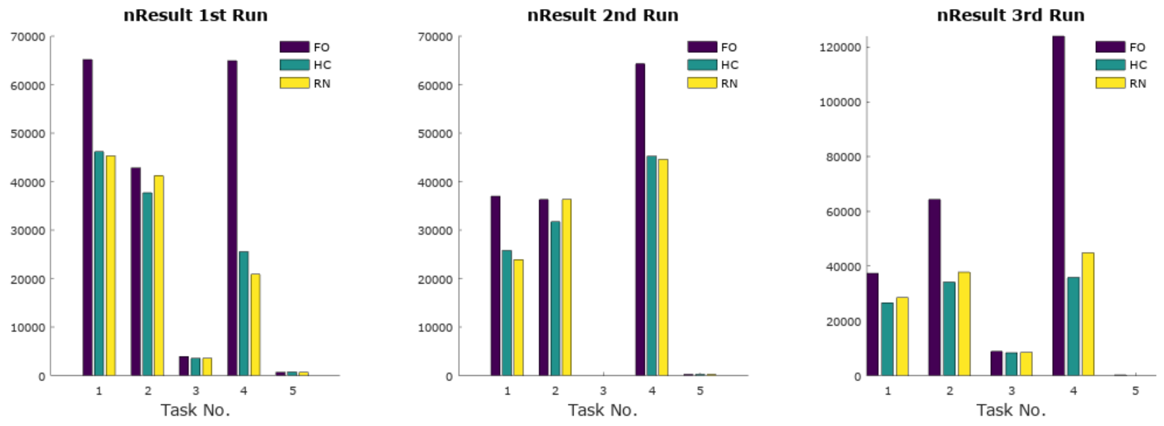


FIGURE 18. The number of delivered result to the destination node of each tasks for three simulation runs of dense scenario (n = 102).

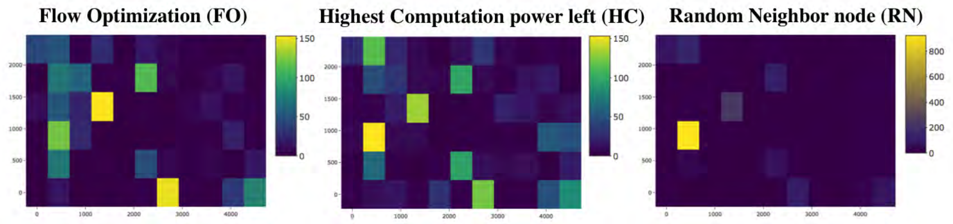


FIGURE 19. Flow volume distribution in dense scenario(N = 102) after 45-minutes.

TABLE 3. Tasks in theoretical scenario.

Task	Input Node	Reduced Ratio	Instruction Count
T_1	fixed S	0.5	1000
T_2	fixed M	0.5	1000
T_3	fixed L	0.1	10000
T_4	mobile M	0.5	1000
T_5	mobile L	0.1	10000

TABLE 4. Average result of dense scenario (n = 102) from three random runs.

Approach	Avg T_p	Avg T_d	Avg *nResult*
Flow Optimization (FO)	1.60	6.41	183511.4
Highest Computation power (HC)	1.14	7.28	107344.4
Random Neighbor (RN)	1.05	6.93	112185.7

To control the total flow affected by each task, we randomly assign a type to each node by considering the evenly total flow generated. Thus, the minimum numbers of each type are 50, 25, 1, 25, and 1, respectively, to generate equal 500 KB$/s$. Destinations for the tasks are random for each running. We set the window time of assignment recalculation and simulation time as 5 and 30 minutes, respectively.

We compare our proposed decision algorithm, flow optimization, (FO) with another two greedy algorithms for a node to choose the offloading node when it reaches the computation capability limit. The first method is to choose the broker with highest computation power left (HC). The second method is to choose the random broker node (RN).

We start with the minimum 102 nodes and randomize destination set for each of three simulation runs. Fig. 18 shows the task results delivered to its specified destination (*nResult*) after 30 minutes. Regardless of the position of destinations, the proposed decision algorithm always provides the higher throughput in term of the number of delivered results.

As presented in Table 4, although the proposed method *FO* has events processed after the compared approaches (see time to be processed (T_p)), it delivers the highest amount of processing results with the fastest delivering time (T_d). In particular, for the dense scenario, where every node produces one input stream and resources are not sufficient, the distributions of incoming flows recorded at the last window after 45-minutes running of all nodes are illustrated as in Fig. 19. Note that, it does not include the self-generated flow. We can observe that our proposed flow optimization (FO) and choosing highest computation power left (HC) can distribute the flow volume across all nodes more efficiently than choosing random broker node (RN). Since there is no sufficient resource until some nodes detect a loop and no choice but do the processing, the nodes always forward the large-volume flows.

Then, we reduce the density of flow volume by increasing the number of nodes to 200, 500 and 1000. Fig. 20 presents the total flow volume after 45 minutes. Our proposed algorithm works best in all cases, especially in the dense scenario.

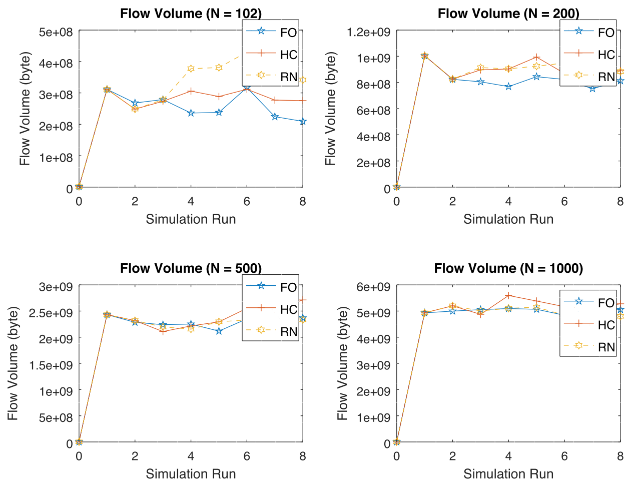


FIGURE 20. Flow volume after 45-minutes.

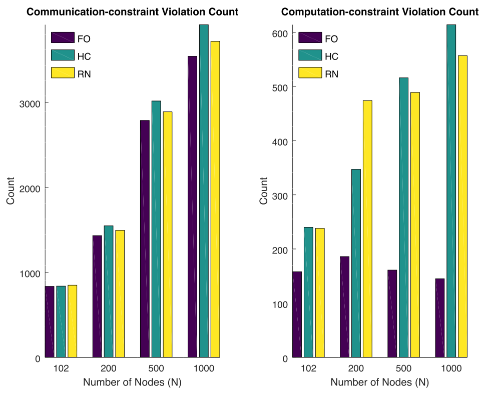


FIGURE 21. Average constraint-violation count of all scenarios.

In the aspect of constraint violations, the proposed approach significantly reduces the violation count for all cases as shown in Fig. 21.

2) NURSING HOME SCENARIO

In this experiment, we use the Java program from Subsection V-B and deploy the plan results in the Scenargie simulation with a smart nursing home scenario consists of three buildings: A, B1, and B2, as depicted in Fig. 22. Input flows are 1.5KB/min from an environmental sensor, 1.5KB/s from a wearable device, and 6.75MB/s from a video streamer. The simulating system runs four requests: (i) General surveillance sending to a local server (ii) Face detection on Building A sending to Internet gateway (iii) Average environmental information sending to an air flow controller for each floor (iv) Heart-rate pattern detection sending to the nearest nurse tablet. In this scenario, we first evaluate our offloading-node selection algorithm (flow-based) to nearest-to-source heuristic approach (hop-based) and all-to-gateway approach (centralized). Results at stable state are summarized in Table 5. The proposed method produces data flow about 7-times lower to the centralized approach. Comparing to the

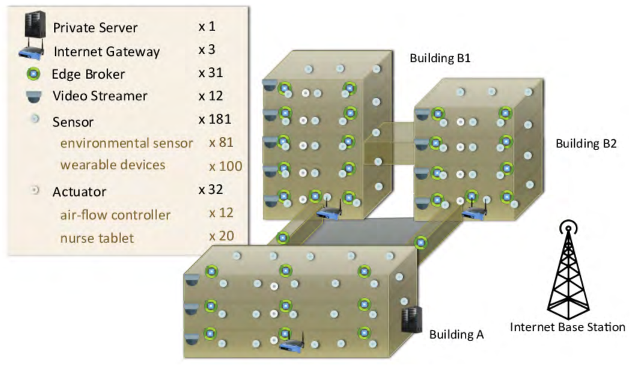


FIGURE 22. Large-scale smart nursing home scenario ($\theta = 25$ K).

TABLE 5. Data flow statistics from large-scale simulation.

Method	#Packets (N)	Bytes (KB)	Loss Rate (%)
Flow based	11,833	17,366.6	3.28
Hop based	11,838	17,373.9	4.54
Centralized	84,501	124,128.3	6.68

hop-based approach, it also outperforms on flow volumes and loss rate. To evaluate our proposed distributed assignment mechanism, we compare its collaborating overhead, a broadcasting message of the node status, to that from the centralized-assigning approach. We summarize results in Table 6, where *cover flood*, *guarantee flood*, and *optimal flood* refer to flooding to all nodes, maximum hop to reach the server, and minimum hop that provides the minimal result as a cover flood, respectively. The total number of packets in the distributed approach is higher than the centralization in general. However, the packet size of the centralized approach is much larger respective to the content numbers. Furthermore, the simulation scenario also shows that the total size of overhead packets from the centralized assigning is larger than the distributed approach with cover flooding.

TABLE 6. Collaborating overhead from large-scale simulation.

Assigning Approach		#Packets (N)	Bytes (KB)
Distributed	Cover flood (Hop=11)	958	92.0
	Guarantee flood (Hop=9)	827	79.4
	Optimal flood (Hop=1)	31	4.0
Centralized	Server pushing	131	139.0

D. REAL-ENVIRONMENT DEPLOYMENT

In this experiment, we deployed the prototype system to six Intel Edison placed in the laboratory with sixteen temperature and humidity sensors, eight tracking sensors, and one camera as depicted in Fig. 23. The video camera flooded twenty of 216 × 144-size frame per second. The temperature and humidity sensors sensed every one minute.

At starting state, five atomic definitions and five composite definitions are loaded into the system. Atomic definitions are *Temp*, *Humid*, *Tracking*, and *Video* for temperature event, humidity event, tracking event, and video-frame

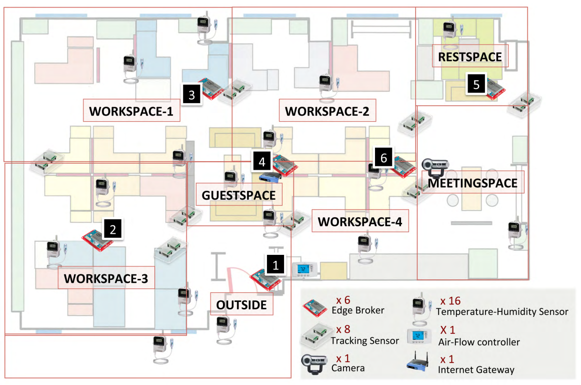


FIGURE 23. Real-environment scenario.

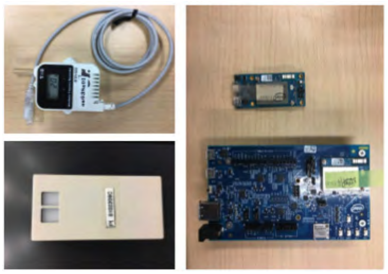


FIGURE 24. Deployed devices: temperature-humidity sensor, tracking sensor, and intel edison module.

TABLE 7. Subscription 1 - out-range average temperature when some people stay.

Subscription	seq 1 from *User*1 to *AirController*
detect	last *PeopleCount2Mins*(*count* > 0) as *PeopleCount* and *AvgTemp5Mins*(*avg* < 20 or *avg* > 25) as *AvgTempOut* within 5 mins from *PeopleCount*
assign	*humid*=*AvgTempOut.avg*, *location*=group.*location*, *timestamp*=*PropleCount.timestamp*
consuming	*AvgTempOut*
group by	*location*

event, respectively. Composite definition are *AvgTemp5Mins*, *AvgHumid5Mins*, *MoveIn*, *MoveOut*, and *PeopleCount2Mins* for event of average value for last 10 minutes in a specific location of temperature, event of average value for last 10 minutes in a specific location for humidity, event when someone comes in a specific location, event when someone get out from a specific location, event to count people in a specific location for every 2 minutes, respectively.

In runtime, there are three subscriptions requested by *user 1* as followings: (1) Out-Range Average Temperature when some people stay (Table 7), (2) Out-Range Average Humidity when some people stay (Table 8), and (3) Face Detection when someone comes in (Table 9).

The results of two-hours run with 10-minutes time-window are concluded in Table 10. We notice that the second and third subscriptions, as well as, referred definition tasks are mainly executed at the destination-nearest broker due to aggregation overhead while the first subscription performs

TABLE 8. Subscription 2 - out-range average humidity when some people stay.

Subscription	seq 2 from *User*1 to *AirController*
detect	last *PeopleCount2Mins*(*count* > 0) as *PeopleCount* and *AvgHumid5Mins*(*avg* < 30 or *avg* > 50) as *AvgHumidOut* within 5 mins from *PeopleCount*
assign	*avg*=*AvgHumidOut.avg*, *location*=group.*location*, *timestamp*=*PropleCount.timestamp*
consuming	*AvgHumidOut*
group by	*location*

TABLE 9. Subscription 3 - face detection when someone comes in.

Subscription	seq 3 from *User*1 to *Gateway*
detect	*MoveIn* and *Video* within 5 mins from *MoveIn*
assign	*face*=FaceDetect(*Video*), *location*=group.*location*, *timestamp*=*MoveIn.timestamp*
consuming	*Video*
group by	*location*

at the source-nearest broker. In the aspect of performance, the summarized results also present that the broker 1 and 2 gain 25% higher communication cost due to aggregation cost and information-exchange overhead. At the same time, communication costs on the rest brokers are reduced at least 60% from the base cost, especially video stream from the broker 6 to 4.

VI. POTENTIAL IMPROVEMENTS

Our proposed approach still has several points that need to be improved. There are three of them that I would like to mention here. Firstly, the current version of our prototype does not allow users to customize the quality of the service. Although users can specify the demands of resource usage limitation, when those demands could not be satisfied, all violations are valued equally. Some edge networks prefer communication loss than overload computation. For example, in the sensor networks on pedestrians of a smart city with very-limited battery power, the monitored information is large but insignificant. Some may have no problem with computation power supply. Still, communication must be reliable such as smart hospital systems. To make a decision, we consider the Multi-Criteria Decision Analysis (MCDA) as one of the potential approaches to the solution [40].

Secondly, as stated in Section IV, there is much real-time information required for computing the distribution plan. The current implementation straightforwardly uses the historical records. In other words, we assume the uniform distribution of arrival. Our system might combine with the arrival workload modeling module [32] to deal with various kind of data distribution. It fits the past-incoming records to the standard distribution and estimates the potential flow volumes beforehand as applied in [31].

Thirdly, even if edgeCEP allows users to extend new functions, it still requires re-building to apply them. Also, the memory of some devices is so limited that cannot even store a big reference function classes. One possible solution

TABLE 10. Average flow volume (bytes per window) from two-hours run of real-environment deployment ($\theta = 0.1$ s, $I = 100$ KB/s).

ID	Tasks	Base Cost	Flow-based Cost				Reduction Ratio
			Output Flow	Aggregation Cost	Overhead	Total	
1	*PeopleCount, *AvgTemp, *AvgHumid Subscription1, Subscription2, Subscription3	9,201.4	659.3	4,255.3	1,988.9	11,312.9	0.8
2	PeopleCount, AvgTemp, AvgHumid, Subscription1	2,950.0	0	796.8	1,987.8	3,826.7	0.8
3	PeopleCount, AvgTemp, AvgHumid, Subscription1	9,010.4	0	756.6	1,989.0	3,869.5	2.3
4	PeopleCount, AvgTemp, AvgHumid, Subscription1	11,995.2	294	772.6	2,092.7	4,396.3	2.7
5	PeopleCount, AvgTemp, AvgHumid, Subscription1	5,996.6	0	756.2	1,988.7	3,847.8	1.6
6	PeopleCount, AvgTemp, AvgHumid, Subscription1	422.5M	294	796.7	1,988.7	4,646.6	90,923.4

* header of aggregation task

is to load an operation code on-the-fly by keeping just only class-owner node and class property. The code-offloading mechanism has been developed in many ways as found in BOINC [41], and Tasklets [42].

VII. CONCLUSION

This paper has introduced a self-organized task distribution framework for module-based event stream processing (*EdgeCEP*). The EdgeCEP is a general complex event processing engine that combines the advantage of an event-dependent specification, along with efficient tuple-based processing by pseudo-source mechanism employing publish-subscribe and content matching techniques. We newly define a supportive event-specification language enabling relational operations. We introduce an optimization problem of the task distribution plan. The proposed framework applies tabu search with a flow-based greedy move to find the sub-optimal solution. The solution computation is periodically executed independently at each node and shared with other necessary information for others to compute the plan. We observe and evaluate the proposed system in many levels and experiments including large-scale simulation and real-environmental deployment. The simulation in the theoretical scenario shows that the proposed flow optimization outperform the trivial algorithms. The average delivering time is almost 1 and 0.5 seconds faster than choosing the highest computation power left and a random broker, respectively, in dense networks. The proposed method can process almost 2-times results more than the others at the end of the simulation. Furthermore, it can reduce total packets 6.6 times from the centralized approach in the practical nursing home scenario. We successfully deployed a prototype engine over an ad-hoc wireless sensor network composed of Intel Edison modules in the real environment. The running result presents decreasing communication cost in general.

REFERENCES

[1] Cisco Internet Business Solutions Group, "The Internet of Things: How the next evolution of the Internet is changing everything," White Paper, Jan. 2011.
[2] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge computing: Vision and challenges," *IEEE Internet Things J.*, vol. 3, no. 5, pp. 637–646, Oct. 2016.
[3] P. G. Lopez *et al.*, "Edge-centric computing: Vision and challenges," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 45, no. 5, pp. 37–42, 2015.
[4] C. Long, Y. Cao, T. Jiang, and Q. Zhang, "Edge computing framework for cooperative video processing in multimedia IoT systems," *IEEE Trans. Multimedia*, vol. 20, no. 5, pp. 1126–1139, May 2017.
[5] G. Li and H.-A. Jacobsen, "Composite subscriptions in content-based publish/subscribe systems," in *Proc. ACM/IFIP/USENIX Int. Conf. Middleware*. New York, NY, USA: Springer-Verlag, 2005, pp. 249–269.
[6] G. Cugola and A. Margara, "RACED: An adaptive middleware for complex event detection," in *Proc. 8th Int. Workshop Adapt. Reflective MIddleware (ARM)*, New York, NY, USA, 2009, pp. 5-1–5-6.
[7] P. R. Pietzuch, B. Shand, and J. Bacon, "A framework for event composition in distributed systems," in *Proc. ACM/IFIP/USENIX Int. Conf. Middleware*. New York, NY, USA: Springer-Verlag, 2003, pp. 62–82.
[8] G. Cugola and A. Margara, "Complex event processing with T-REX," *J. Syst. Softw.*, vol. 85, no. 8, pp. 1709–1728, Aug. 2012, doi: 10.1016/j.jss.2012.03.056.
[9] (2015). *StreamBase*. [Online]. Available: http://www.streambase.com
[10] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica, "Discretized streams: An efficient and fault-tolerant model for stream processing on large clusters," in *Proc. 4th USENIX Conf. Hot Topics Cloud Comput. (HotCloud)*, Berkeley, CA, USA, 2012, p. 10. [Online]. Available: http://dl.acm.org/citation.cfm?id=2342763.2342773
[11] *Storm: Distributed Realtime Computation System*. Accessed: Jul. 6, 2017. [Online]. Available: http://storm.apache.org/
[12] *WSO2*. Accessed: Jul. 6, 2017. [Online]. Available: http://wso2.com/
[13] A. G. De Prado, G. Ortiz, and J. Boubeta-Puig, "CARED-SOA: A context-aware event-driven service-oriented Architecture," *IEEE Access*, vol. 5, pp. 4646–4663, 2017.
[14] Y. Tian, E. Ekici, and F. Ozguner, "Energy-constrained task mapping and scheduling in wireless sensor networks," in *Proc. IEEE Int. Conf. Mobile Adhoc Sensor Syst. Conf.*, Nov. 2005, pp. 8 and 218.
[15] Freifunk. (Jan. 2017). *B.A.T.M.A.N. Advanced*. [Online]. Available: https://www.open-mesh.org/projects/batman-adv/
[16] G. Cugola and A. Margara, "Processing flows of information: From data stream to complex event processing," *ACM Comput. Surv.*, vol. 44, no. 3, pp. 15:1–15:62, Jun. 2012.
[17] A. Arasu, S. Babu, and J. Widom, "The CQL continuous query language: Semantic foundations and query execution," *VLDB J.*, vol. 15, no. 2, pp. 121–142, Jun. 2006.
[18] A. Akbar, A. Khan, F. Carrez, and K. Moessner, "Predictive analytics for complex IoT data streams," *IEEE Internet Things J.*, vol. 4, no. 5, pp. 1571–1582, Oct. 2017.
[19] M. N. Alkhomsan, M. A. Hossain, S. M. M. Rahman, and M. Masud, "Situation awareness in ambient assisted living for smart healthcare," *IEEE Access*, vol. 5, pp. 20716–20725, 2017.
[20] Y. Sun, T.-Y. Wu, G. Zhao, and M. Guizani, "Efficient rule engine for smart building systems," *IEEE Trans. Comput.*, vol. 64, no. 6, pp. 1658–1669, Jun. 2015.
[21] G. Li, V. Muthusamy, and H.-A. Jacobsen, "Adaptive content-based routing in general overlay topologies," in *Proc. Middleware ACM/IFIP/USENIX 9th Int. Middleware Conf.* Berlin, Germany: Springer, 2008, pp. 1–21.
[22] S. Wasserkrug, A. Gal, O. Etzion, and Y. Turchin, "Efficient processing of uncertain events in rule-based systems," *IEEE Trans. Knowl. Data Eng.*, vol. 24, no. 1, pp. 45–58, Jan. 2012.
[23] Y. Shoham and K. Leyton-Brown, *Multiagent Systems: Algorithmic, Game-Theoretic, and Logical Foundations*. New York, NY, USA: Cambridge Univ. Press, 2008.
[24] A. Akbar *et al.*, "Real-time probabilistic data fusion for large-scale IoT applications," *IEEE Access*, vol. 6, pp. 10015–10027, 2018.
[25] G. Cugola and A. Margara, "Deployment strategies for distributed complex event processing," *Computing*, vol. 95, no. 2, pp. 129–156, 2013.

[26] P. Pietzuch, J. Ledlie, J. Shneidman, M. Roussopoulos, M. Welsh, and M. Seltzer, "Network-aware operator placement for stream-processing systems," in *Proc. 22nd Int. Conf. Data Eng. (ICDE)*, Washington, DC, USA, 2006, p. 49, doi: 10.1109/ICDE.2006.105.
[27] M. H. A. Awadalla, "Task mapping and scheduling in wireless sensor networks," *IAENG Int. J. Comput. Sci.*, vol. 40, no. 4, pp. 257–265, 2013.
[28] H. Park and J. W. Lee, "Task assignment in wireless sensor networks via task decomposition," *Inf. Technol. Control*, vol. 41, no. 4, pp. 340–348, 2012.
[29] A. T. Zimmerman, J. P. Lynch, and F. T. Ferrese, "Market-based computational task assignment within autonomous wireless sensor networks," in *Proc. IEEE Int. Conf. Electro/Inf. Technol.*, Jun. 2009, pp. 23–28.
[30] B. Dieber, L. Esterle, and B. Rinner, "Distributed resource-aware task assignment for complex monitoring scenarios in visual sensor networks," in *Proc. 6th Int. Conf. Distrib. Smart Cameras (ICDSC)*, Oct. 2012, pp. 1–6.
[31] R. Mayer, B. Koldehofe, and K. Rothermel, "Predictable low-latency event detection with parallel complex event processing," *IEEE Internet Things J.*, vol. 2, no. 4, pp. 274–286, Aug. 2015.
[32] D. G. Feitelson, *Workload Modeling for Computer Systems Performance Evaluation*, 1st ed. New York, NY, USA: Cambridge Univ. Press, 2015.
[33] N. Zacheilas, V. Kalogeraki, Y. Nikolakopoulos, V. Gulisano, M. Papatriantafilou, and P. Tsigas, "Maximizing determinism in stream processing under latency constraints," in *Proc. 11th ACM Int. Conf. Distrib. Event-based Syst. (DEBS)*, New York, NY, USA, 2017, pp. 112–123, doi: 10.1145/3093742.3093921.
[34] L. Fischer and A. Bernstein, "Workload scheduling in distributed stream processors using graph partitioning," in *Proc. IEEE Int. Conf. Big Data (Big Data)*, Oct. 2015, pp. 124–133.
[35] Y. Nakamura, T. Mizumoto, H. Suwa, Y. Arakawa, H. Yamaguchi, and K. Yasumoto, "*In-situ* resource provisioning with adaptive scale-out for regional IoT services," in *Proc. 3rd ACM/IEEE Symp. Edge Comput. (SEC)*, Oct. 2018, pp. 203–213.
[36] S. Choochotkaew, H. Yamaguchi, T. Higashino, M. Shibuya, and T. Hasegawa, "EdgeCEP: Fully-distributed complex event processing on IoT edges," in *Proc. 13th Int. Conf. Distrib. Comput. Sensor Syst. (DCOSS)*, Jun. 2017, pp. 121–129.
[37] G. Cugola and A. Margara, "TESLA: A formally defined event specification language," in *Proc. 4th ACM Int. Conf. Distrib. Event-Based Syst. (DEBS)*, New York, NY, USA, 2010, pp. 50–61.
[38] S. Martello and P. Toth, *Knapsack Problems: Algorithms and Computer Implementations* (Wiley Series in Discrete Mathematics and Optimization). Hoboken, NJ, USA: Wiley, 1990.
[39] E. Friedman-Hill. (2007). *Jess, The Rule Engine for the Java Platform*. [Online]. Available: http://herzberg.ca.sandia.gov
[40] G. Britain, "Multi-criteria analysis: A manual," Dept. Communities Local Government, London, U.K., 2009.
[41] D. P. Anderson, "BOINC: A system for public-resource computing and storage," in *Proc. 5th IEEE/ACM Int. Workshop Grid Comput.*, Nov. 2004, pp. 4–10.
[42] D. Schafer, J. Edinger, J. M. Paluska, S. VanSyckel, and C. Becker, "Tasklets: 'Better than best-effort' computing," in *Proc. 25th Int. Conf. Comput. Commun. Netw. (ICCCN)*, Aug. 2016, pp. 1–11.

SUNYANAN CHOOCHOTKAEW received the B.Eng. degree in computer engineering from Chulalongkorn University, Thailand, in 2014, and the M.E. degree in information and computer sciences from Osaka University, Japan, in 2017, where she is currently pursuing the Ph.D. degree with the Mobile Computing Laboratory, Graduate School of Information Science and Technology, under the supervision of Prof. T. Higashino.

HIROZUMI YAMAGUCHI received the B.E., M.E., and Ph.D. degrees in information and computer sciences from Osaka University, Japan, in 1994, 1996, and 1998, respectively. He is currently an Associate Professor with Osaka University. His current research interests include design, development, modeling, and simulation of mobile and wireless networks and applications. He is a member of the IEEE.

TERUO HIGASHINO received the B.S., M.S., and Ph.D. degrees in information and computer sciences from Osaka University, Japan, in 1979, 1981, and 1984, respectively. He joined the faculty of Osaka University, in 1984. Since 2002, he has been a Professor with the Graduate School of Information Science and Technology, Osaka University. His current research interests include design and analysis of distributed systems, communication protocol, and mobile computing. He is a Senior Member of the IEEE, and a Fellow of IPSJ.