On-demand Task Selecting and Offloading Method in Self-organizing Distributed Stream Processing

Sunyanan Choochotkaew[1,a)] Hirozumi Yamaguchi[1,b)] Teruo Higashino[1,c)]

Abstract: In this paper, we emphasize challenging task distribution on the self-organized systems and propose two-fold heuristic mechanisms called QoS-oriented selection and on-demand offloading. The first-fold is a long-term decision from performance and QoS-requirement factor coupling with a delay estimation technique. The second-fold is a short-term decision utilizing a queuing theory to determine overloading state. In the evaluation part, there are proof-of-concept experiments for the selection method and delay estimation model and a real-deployment showing that processing results corresponding to task-specified QoS requirement and outperformance over the comparing approach.

1. Introduction

Integrating intelligence in real-time promises further benefits on many kinds of applications from macro-scale (e.g., city surveillance, transportation system) down to personal comforts (e.g., home sensing and appliance control). To serve as a basis, stream processing technology has been continuously studied and developed for decades to overcome the challenge of a real-time requirement. The stream processing engine is a tool to extract and transform the physically sensing data into desirable/ready-to-use knowledge. Beyond the concept of stream processing, Complex Event Processing (CEP) ordinarily follows through the process from sensing at sensors till activating the automated control actions at actuators according to user-defined requests.

Most engines enhance performance by increasing the number of processing units distributed in-between to execute the generated tasks from collected sensor streams. Even if the centralized-like approach on the cloud is illimitable to scale out, there are also a few issues that need more concern such as budget, security, latency, and connectivity. The edge computing is a welcomed solution to deal with remote-processing problems [12][27]. With powerful local servers, it can also parallelize the task generating and distributing part like the system proposed in [7]. In spite of that, an additional cost of local servers and their maintenance is a considerable tradeoff. The way to cut down such a cost is to draw out powerfulness from non-dedicated devices. For instance, in a small collaborative community, the community may want to automatically recognize activity in the public park from video frames and submit to alarm devices. An intelligent security system with a few smart cameras and smart alarms may be sustainable with only collaborations from nearby-established such as workplace computers and passing-by devices such as mobile phones. Furthermore, the video sources do not limit to only available cameras but allow any community members to provide their data in the close-up view or blind spot.

Without any known dedicated device, the self-organizing and fully-distributed system requires a collaborative mechanism to run independently on each participating device for serving the global request set. In ref.[4], an agent in the multi-agent system applies divide and conquer for exploring the sensing data in structural topology to provide event learning service in a self-organizing manner. It can migrate the tasks when finding unreliability of missing links or nodes by a hybrid approach of random and attractive walk behavior with pre-defined routing. Nevertheless, they work regardless of task variety. A different task may focus on a different quality attribute. For example, an urgent such as falling-detecting task requires a low latency while a traffic-reporting task requires high reliability. To the best of our knowledge, there was no discussion on the task-specific QoS on multi-purpose processing systems in a self-organizing context.

In this paper, we introduce a self-organized scheme on multi-purpose stream processing systems tearing off the task-collaborating mechanism into two phases. The first phase, named QoS-oriented recipe selection, is to choose a set of tasks to serve. The second phase, named on-demand offloading, is to hand the overloaded work to a potential neighbor. The QoS-oriented selection algorithm finds the consensus to serve all tasks concerning task-specific QoS over device performance and resource constraints. The on-demand offloading algorithm uses regression technique and queuing theory to determine an overloading condition. In the evaluation, firstly, we conduct proof-of-concept experiments for QoS-oriented task selection algorithm and processing time estimation model. Secondly, we deploy the proposed system on real-world devices and evaluate its performance with the standard policy. The results show that the proposed method provides the best-compliance to the task-specific requirements.

1 Information Networking Department, Graduate School of Information Science and Technology, Osaka University
a) sunya-ch@ist.osaka-u.ac.jp
b) h-yamagu@ist.osaka-u.ac.jp
c) higashino@ist.osaka-u.ac.jp

The rest of this paper is organized as follows. Section 2 states related works and a summary of paper contributions. Next, section 3 gives a background of the proposed system. Section 4 then describes the proposed mechanism in detail. After that, section 5 presents the evaluation results. Lastly, section 6 provides the paper conclusion.

2. Related Work

There have been several studies towards distributed stream processing encountering the challenge of continuous-but-irregular demand over heterogeneity of distributed processing units [8]. Among those, the most engaging problem is task distribution as it has a direct influence on service quality. This may be referred by other interchangeable terms such as operator/content placement in [28][24][25], task mapping in [3], and task assignment in [13][16], and task allocation in [6]. The common point is to map a set of tasks to a set of processing units. Tasks may refer to the operator and the data input or either of them. Some propose a heuristic approach owing to simplicity advantage [16][2] while some consider applying more complicated approximation approach [7]. In the quality aspect, most proposed systems only provide a best-effort solution rather than a requirement-guarantee solution. To assure the satisfying quality, some add a negotiation mechanism and violation constraints, for example, in [14].

The nature of the task in stream processing usually separates input data apart from the operator since the operator is long-lasting while the input data is always varied. For scaling-out the distribution performance in complicated processing request, the stream-processing systems mostly apply a decomposition technique [13] and perform operator placement. The decomposition technique is to decompose the operator part in the recipe into multiple operators connecting in sequence represented by a DAG graph. An output from the previous sequence is an input of the connecting operator, namely, temporary tables in the tuple-based model and composite event in the event-based model. For utmost power utilization, the decomposed component in one graph often allows the reuse. The operator placement is to map the task graph into physical processing topology [3]. With central synchronization, *RECEP* introduces selection layer and formulates Quality of Result estimation function as a key of distribution solution [23]. In ref [17], the geographic address has been introduced and auto-configured by local matching service.

Since the operator is placed once and runs forever, a processing unit may face the overloading situation from unexpected loads of continuous streams, and that might cause an unacceptable latency. Ref. [19] determines arrival distribution over time to predict future workload [10], in the meantime, performs operator profiling to calculate minimal parallel degree. With accurate prediction, we may be able to distribute the load properly in advance. However, on-the-fly operators and self-organized distributing decision making such detailed profiling more difficult. Another solution is to detect the overloading situation and perform task offloading. We consider that this is an applicable concept to task migration from the dying state. In ref [13], when a device detects the dying state from battery level, it performs a probing mechanism to find the neighbor with minimum cost to continue the task. Nevertheless, such passive probing incurs unsatisfying interruption on volunteering participants since most of responding participants are not selected. In ref.[15], the authors introduce *gradient model* to inform a current load level of each device. In ref.[21], the operator (service) placement is adaptively scaled out to the nearby cluster according to overloading demand. Despite that, there is no overloading detection directly to the service time guarantee.

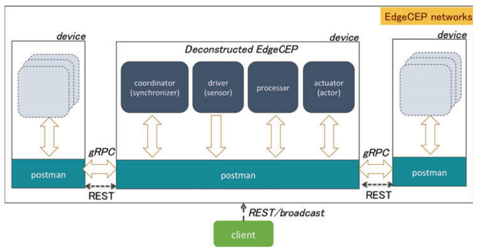


Fig. 1: μEdgeCEP: Modularized EdgeCEP system

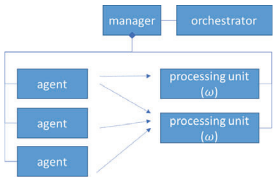


Fig. 2: Processor Class Relation

To summarize the idea, in this paper, we emphasize the importance of a self-organizing mechanism for distributed stream processing on non-dedicated networks of devices. We take minimum requirement of neighbor interruption into consideration for designing the collaborative mechanism. Throughout this paper, we reconsider a placement problem of event-decomposition as recipe selection (interchangeably with task selection). The following are our summarized contributions:

(1) we propose a scoring method of task-specified QoS requirement over device performance to be a key factor
(2) we introduce generic-granularity model to predict processing time for comparing time-guarantee constraint
(3) our task selection uses only long-term selection decision from neighbors to find the system consensus
(4) we apply a queuing theory to determine overloading state and offloading decision
(5) we conduct proof-of-concept experiments for our selection method and processing-time prediction model
(6) we develop system prototype and show our superiority in terms of QoS-corresponding result and processing delay

3. Background and Modularized EdgeCEP

Nowadays, the term stream processing is interchangeable with complex event processing as they both refer to the continuous operation to execute processing request flow from data origins to subscribing destinations defined by system users. In the proposed system, we use **Event** model for specifying the flow origin and destination instead of a fixed stream identifier. As a result, users can define a request flow once and apply to all relevant producers and consumers. Any producers or consumers can leave or join the networks transparently to users. Devices can generate an event from two origins: direct driver (called connector in Kafka[22]) and processing content defined by users. Throughout this paper, we use the term **Recipe** to refer to an event specification and use the term **Task** to refer to a job to follows a user-specified processing content. The driver interprets data such as sensing or logging values in a designated sampling window to an event according to an atomic event recipe, which contains only name and attributes. The processing content produces a new type of events from the other previously-defined events. In a complex event field, the event from the driver is called *atomic event*, and that from the processing content is called *composite event*. We implement the proposed distributing mechanism on our developed EdgeCEP system [5] in modular version, called *μEdgeCEP*.

EdgeCEP is a fully-distributed complex event processing for driving on-the-fly recipes over networks of non-dedicated edge devices. Edge devices may have any functionality of producers, consumers, and processors. It has been first-time introduced with monolithic architecture in [5]. Nevertheless, it might be too heavy to put the processing burden to some limited or busy sensors and actuators and even be worthless to keep processing-dependent libraries and tools on those incapable devices. To avoid such inefficient deployment, we decouple those functionalities from the base service that is self-organizing communication, called *postman*, as shown in Fig. 1. In the proposed system, devices flexibly install only preferable functionalities. For instance, the devices with connection to sensing source or actuators may install only driver or actuator service in case that they do not have enough capability of processing. The devices with long-lasting online hours may install coordinator service to help synchronize recipes over the networks regardless of processing power. We note that the system must have at least one coordinator service run at any moment to keep all devices synchronizing.

A recipe in *μEdgeCEP* contains processing content set, output-event constructor, and actuator function call. The processing content set can have multiple content cases sharing the same recipe name and output handling. For example, we may specify falling one way from processing on smartwatches, and another way from processing on cameras. A content case is a processing specification comprising of an operator, input collection specification, grouping policy, and consumption policy, found the description in [5]. In this paper, we mainly focus on the operator element in the processing content. The operator has information on processing kind, processing-flow graph, input definition, and output definition. We apply TensorFlow[1] packed module to handle processing-flow graph and use the pre-defined processing kind to designate the granularity of processing-time prediction model.

The processing flow starts from an event generated from *driver* service, and forwarded to *postman* service. The filterer component at *postman* service then performs fast-mapping of the event to the related recipe name and content case identifier according to the input collection specification in the recipe, and deliver the filtred event and mapping results to the recipe agents in the *processor* service. The processor service comprises of four classes. *Manager* class provides GRPC service to handle messages from the postman service. *Orchestrator* class acts as a coordinator of processor services in different devices. *Agent* and *Processing unit* classes are parallel threads. One agent is responsible for one task

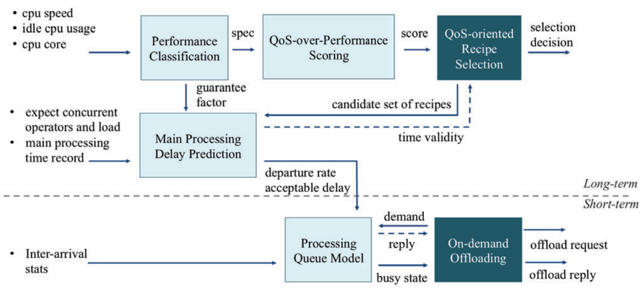


Fig. 3: On-demand Task Selecting and Offloading Method

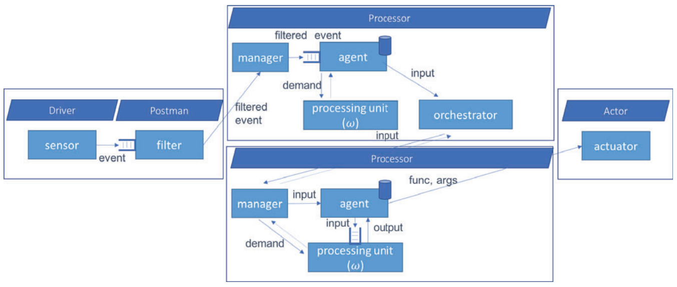


Fig. 4: Offloading Procedure

(τ) starting from collecting till handling the processing output specified in a specific recipe. One processing unit is responsible for executing a specific operator (ω) on collected input data from agent class and returning output to the callback function of that individual inputting. The class relations are depicted in Fig. 2. Since multiple recipes can share an operator (i.e., reuse), a processing unit may receive input from more than one agent. On the other hand, as multiple content cases can apply a different operator, one agent may connect to more than one processing unit. The callback function is to deliver the output as arguments to the *actor* service specified in the recipe.

4. On-demand Task Selecting and Offloading

The idea of an on-demand approach is to act only when finding it necessary. The goal is to reduce the network traffic as well as communication cost and latency while preserving a recipe-focus QoS. To achieve that, we propose a two-fold mechanism integrated into the *μEdgeCEP*: QoS-oriented task selection and on-demand offloading decision-making. The method flow is summarized in Fig. 3. For the first fold, QoS-oriented recipe selection, orchestrator performs the recipe selection to limit the agent and processing units threads to serve only a demanding set of recipes. A preference score of each recipe according to its specified weight against QoS attributes is from QoS-over-Performance scoring module, which further referring the device spec from the performance classification module. To select a recipe into the demanding set, it considers not only a high preference score but also the memory and time constraints. In particular, the processing delay prediction module provides a validity check over time-guarantee constraint. For the second fold, on-demand offloading, processing units hand its overloading work to neighbors when it meets a dying state or overloading condition during task processing. In this paper, we mainly contribute in the latter case where the caller processing unit determines an overloading condition, and the callee checks its availability to accept the offloaded tasks

```
Data: $\tau^+, \Upsilon'$
Result: $\Upsilon$, $score$, $Q$
$S$ = Pool ($\tau^+$)
$S_r$ = RequiredPool ($S, \Upsilon'$)
$S_{copy}$ = copy($S$)
$\Upsilon = \phi, \mu = \phi, \overline{d} = \phi, score = 0$
if $S_r = \phi$ then
    $S_r = S$, $selector$ =MaxSelector ()
else
    if is_competitive ($S_r, \Upsilon'$) then
        $selector$ = OpportunisticSelector ()
    else
        $selector$ = MaxSelector ()
    end
end
while $S_r \neq \phi$ do
    $\upsilon' \leftarrow selector$.select ($\Upsilon, S_r$)
    $\Upsilon' \leftarrow \Upsilon \bigcup \{\upsilon'\}$
    $valid$ = mem_check ($\Upsilon'$)
    if $valid$ then
        $valid, \mu', \overline{d}' \leftarrow$delay_predict ($\Upsilon', \mu, \overline{d}$)
        if $valid$ then
            $\Upsilon \leftarrow \Upsilon', \mu \leftarrow \mu', \overline{d} \leftarrow \overline{d}'$
            add Score ($S_{copy}, \upsilon'$) to $score$
        end
    end
    remove $\upsilon'$ from $S_r$ and $S$
    if $S_r = \phi$ then
        $S_r = S$, $selector$ =MaxSelector ()
    end
end
$Q \leftarrow$Queue ($\mu, \overline{d}$)
```

Algorithm 1: Pseudo code of QoS-oriented Recipe Selection

from the queue model. The queue model obtains the predicted departure rate and acceptable delay from the processing delay prediction module, and estimate arrival rate from the inter-arrival statistic. The details of the modules on the flow are as below:

4.1 Performance Classification

In general, the performance indicators are clock speed and core number. The CCM article, titled "A Quad-Core at 2.66Ghz or a Dual Core at 3.33GHz?", states the relation between both indicators to the gaming performance metric (frames per seconds, FPS) from a benchmark in magazine [18]. We fit those observations to a linear regression model to predict processor performance from clock speed and its available cores at idle state:

$$\text{perf.} = x_1\text{Speed}_{clock} + x_2\text{Core}_{available} \qquad (1)$$

The available core at idle state is $\frac{(100-\%\overline{CPU}_{idle})}{100} \times \text{Core}_{total}$. With system-defined threshold (β_c) of performance values, processors fall into three classes: high, medium, and low.

4.2 Main Processing Delay Prediction

According to a finding from our experiments in section 5, the number, and kind of processes running in parallel also affect each other processing time added to its workload. To provide a flexible tradeoff between prediction accuracy and practicality, we categorize arbitrary operators into a predefined processing kind with some co-features. The kind may be trivial like simple representative complexities (e.g., linear, polynomial, and exponential) or derived from learning technique (e.g., k-means clustering). The higher number of kinds may provide better fitting of the model. However, it requires more observation to train on each kind. Let Π be a predefined set of kinds, we model the main delay of a processing operator kind $\pi \in \Pi$ from the total number and average load amount of all processing operator running in parallel, denoted by n'_π and l'_π, respectively. Suppose $\forall_{\pi'\in\Pi}\ a_{\pi\pi'}$, $b_{\pi\pi'}$, and ε_π for parameters from fitting the training data to the linear model of π-kind delay, the general prediction function of π-kind delay is:

$$d(\pi) = \Sigma_{\pi'\in\Pi}(a_{\pi\pi'}n_{\pi'} + b_{\pi\pi'}l_{\pi'}) + \varepsilon_\pi \qquad (2)$$

An acceptable delay for each operator kind is its base delay multiplied by processor guarantee factor (α_c). The base delay, denoted by $d_{base}(\pi)$, is the delay when there is no other parallel processes, $n_{\pi'} = 1$ if $\pi' = \pi$ otherwise $n_{\pi'} = 0$. The processor guarantee factor is a system-defined parameter to compromise the processing overhead added to the base processing time. This factor may be different depending on the performance class. For example, the high-class processor may provide 1.5-times guarantee while the low-class processor may guarantee with 2.5-times over the best-effort latency.

4.3 QoS-over-Performance Scoring

This module is to give a score of the considering recipe (τ) according to its specified QoS weights and processor spec. The higher score infers a higher chance to be selected and handled by an agent. In this paper, we focus on only commonly-considered QoS attributes (θ) comprising of efficiency (e), reliability (r), and availability (a) and denote processors performance class (c) as h, m, l for high, medium, and low class, respectively. The weight on efficiency, reliability, and availability are represented by w_e, w_r, and w_a, respectively. The total weight ($w_e + w_r + w_a$) must be 1.0. For reliability and availability, any class of processors reflects the same preference, denoted by p_0. Meanwhile, efficiency causes a different preference in a different class. We denote the efficiency advantage for high, medium, and low classes by γ_h, γ_m, and γ_l, respectively. To keep balance advantage of all attributes, the total advantage of the efficiency attribute on all classes must be $\gamma_h + \gamma_m + \gamma_l = 3 \times p_0$. For simplicity, we usually consider p_0 as 1. The requirement of replications is directly from the weight of reliability, w_r. The recipe with reliability weight w_r will have $\lceil w_r \times M \rceil$ replications. As the replication must be more than 0, the reliability weight must be positive, $w_r > 0$. To suppress the advantage of replication, the preference over reliability decreases upon the already-selected count. We consider only the selections from neighbors with higher performance, denoted by Υ'_+, have stability and domination. Let $|\tau \in \upsilon^+_{neigh}|$ be the number of times that the considering recipe selected by higher-performance neighbors, we compute the score of recipes at the processor with class $c \in \{h, m, l\}$ by the following equation:

$$score_c = (w_e\gamma_c + w_a) \times M + w_r \times max(1, M - |\tau \in \Upsilon^+_{neigh}|) \quad (3)$$

4.4 Processing Queue Model

Each processing unit owns its queue to serve processing request from referring agents in order. According to main processing delay prediction, we can estimate the departure rate of each processing unit queue as $\mu = 1/d(\pi)$ and set the acceptable delay to $\overline{d} = \alpha_c d_{base}(\pi)$. During the running time, a processing unit records inter-arrival time when a new input added to find the average arrival rate, denoted by λ. We always consider departure distribution as deterministic. On the other hand, the arrival distribution will be deterministic if and only if all referring recipes are deterministic, modeled by D/D/1. Otherwise, it is stochastic from merging property of Poisson processes [11], modeled by M/D/1. Where the queue utilization is denoted by $\rho = \lambda/\mu$, the mean response time of processing unit with queue model is:

$$d_q = d(\pi) + d_{wait} = \begin{cases} \frac{1}{\mu} + \frac{\rho}{2\mu(1-\rho)} & ; \text{Q=M/D/1 and } \rho \leq 1 \\ \frac{1}{\mu} & ; \text{Q=D/D/1 and } \rho \leq 1 \\ \infty & ; \rho > 1 \end{cases} \quad (4)$$

4.5 QoS-oriented Recipe Selection

The recipe selection is a long-term decision done at the processor service on each device periodically. The service will advertise its selection only when there is an update on its selection decision or when it receives a message from a new or outdated neighbor. Hence, the source devices can distribute the filtered input to the processors according to those advertisements. Once the processor leaves the networks, the source devices then remove that processor from the distributing list. The selection method is described by pseudo-code in Algorithm 1. Firstly, we construct the complete pool (S) from a set of active recipes (τ^+) accordingly to the score from equation 3. Then, we construct the required pool (S_r) from deducting the complete pool with existing replications selected by neighbors (Υ'). To converge global decision to a consensus that is all recipes are selected by its requested number of replicas while considering suitability between selected set and device performance, we provide two selection policy: opportunistic and maximum selection. The opportunistic selection is for competitive cases where more than one device with the same spec fight for the same recipe. Otherwise, processors apply the maximum selection policy for obtaining the maximum score.

The selection gives priority to the required pool. If there is no more in-require recipe, it will continue with the rest recipes in the full pool. For each new selection (υ') and corresponding candidate set (Υ'), the memory usage is early computed and validated from the accumulated operator module size. If the newly-selected recipe does not violate the memory capacity, it will next validate the time-guarantee constraint in the main-processing delay prediction module. At the same time, the prediction module will compute the departure rate (μ) and acceptable delay ($\overline{d}$) of each referred processing operator. If all constraints are satisfied, it will add the newly-selected recipe to the demanding set (Υ) and update departure rates, acceptable delays, and scores. Lastly, it will construct the queue model from the final values of departure rates and acceptable delays for the referred processing units.

4.6 On-demand Offloading

The offloading in the *μEdgeCEP* system is conceptually illustrated in Fig. 5 with dot lines. When the processor faces overload condition, it will offload the ready input to its neighbor processor through postman connection. The offload-callee processors will handle the rest part (output handling and delivering) without returning anything to the offload caller. Fig. 4 presents the offloading procedure. In particular, it is a short-term decision by the cooperation of agents and processing units. When the input is ready, the agent will submit the current demand (i.e., the number of input waiting in the queue), denoted by χ, and the processing unit will compute probability to accept the new amount of demand by the following equation corresponding to its queue model under acceptable time constraint ($\overline{d}$) Firstly, we compute the maximum arrival number (λ_{max}) during time period (T) from setting $d_q = \overline{d}$ in equation 4, presented by equation 5.

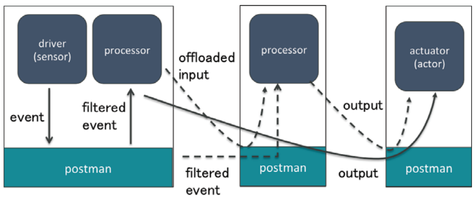


Fig. 5: Event Processing Flow

$$\lambda_{max} = \begin{cases} \frac{2\mu(\mu\overline{d}-1)}{(2\mu\overline{d}-1)} & ; \text{Q=M/D/1} \\ \mu & ; \text{Q=D/D/1} \end{cases} \quad (5)$$

Accordingly, the acceptance probability is the probability to have χ inputs in the system with the maximum arrival rate (λ_{max}):

$$P(\chi|\lambda_{max}) = \begin{cases} [\frac{\lambda_{max}}{\mu}]^n[1 - \frac{\lambda_{max}}{\mu}] & ; \text{Q=M/D/1} \\ 1.0 & ; \text{Q=D/D/1 and } \chi \leq 1 \\ 0 & ; \text{otherwise} \end{cases} \quad (6)$$

If the processing unit rejects the request, the agent will ask orchestrator to offload the input content to other processors. When the manager of a callee processor receives the offload request, it will check the busyness ($d_q \geq \overline{d}$) of its active agent and processing unit. If both are available, the callee agent will directly enqueue the input to its corresponding processing unit.

5. Experiment and Evaluation

In this section, we discuss results from a couple of preliminary experiments and evaluate our proposed systems coupling with different offloading approaches. The preliminary experiments run on a logical environment setting for giving the proof of concept on the proposed recipe selection algorithm and processing delay modeling factors. For system evaluation, we deployed our prototype developed in python to six Intel Edisons, four virtual machines on macOS, host machine, and one portable laptop.

5.1 QoS-Oriented Recipe Selection

To study our proposed recipe selection algorithm, we assume three types of recipes: efficiency-intensive, reliability-intensive, and availability-intensive The efficiency advantages on high, medium, and low processor classes are 2.4, 0.6, and 0, respectively. The maximum replication (M) is 10. Supposing every task costs the same main-processing delay, the high-class, medium-class, and low-class processors respectively spend 0.02s, 0.05s, and 0.1s to finish the task. To preserve the same utilization over

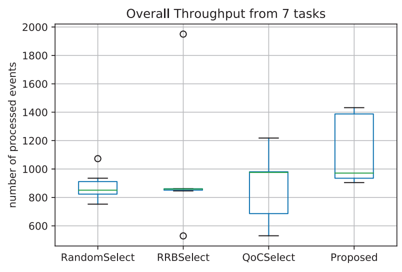


Fig. 6: Summary of QoS-oriented throughput from all recipes

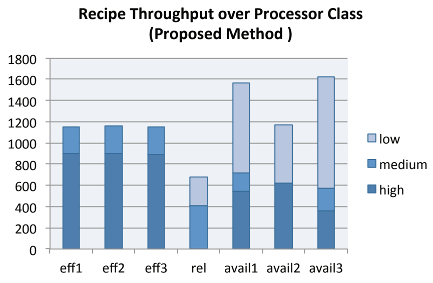
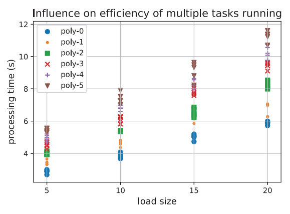
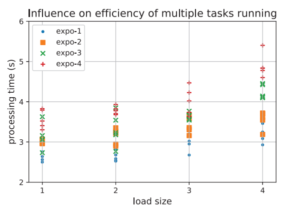


Fig. 7: QoS-oriented Throughput



(a) Varied Polynomial operator



(b) Varied Exponential operator

Fig. 8: Effects of Parallel Processing

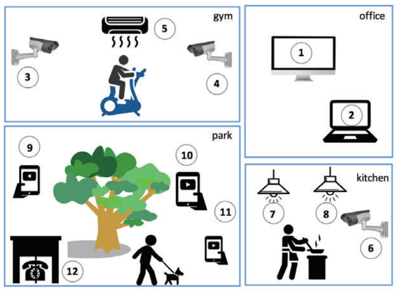


Fig. 9: Assuming Scenario

the time window, the maximum capacity of memory units for each class are 5, 2, and 1.

Since one reliability-intensive recipe requires at least ten processors, we run two high-class, five medium-class, and ten low-class processors to achieve the minimum requirement of replications and balance the total processing power of each class. Accordingly, the densest scenario is to fill with recipes from efficiency-intensive and availability-intensive types equally (i.e., three for each). To summarise, there is a memory capacity for thirty recipe-operating units from seventeen processors, and there are sixteen memory units required (i.e., 10 memory units for one reliability-intensive recipe, 6 memory units for each of the rest six recipes). As a recipe cannot reserve more than one memory unit in the same device, the maximum reservation (i.e., selectable count) equals to the number of processors. We set the sampling rate of each input stream to a minimum value of delay (0.02s) and total testing time to 60 seconds. The compared approaches are random, round-robin (RRB), and QoC selections. QoC selection follows the explicit policy. It firstly lets the high-class processors select the efficiency-intensive tasks, then, reserves the memory units for the reliability-intensive task, and, lastly, gives the left memory units to availability-intensive tasks. For the random approach and our proposed method, we perform 10-times running samples and use an average value due to opportunistic selection.

From the result presented in Fig. 6, the proposed method yields the highest total throughput, specifically, 29.4%, 17.6%, and 31.3% higher from random, RRB, and QoC methods. Furthermore, our method reflects the QoS requirement satisfaction, as observed in the distribution of throughput from a sample run, presented in Fig. 7. The efficiency-intensive recipes mostly run on high-class processors, and all processor classes select any of the availability-intensive recipes.

5.2 Processing Delay Prediction Model

As the processing agent threads sharing the resources on the same machine, the number and complexity of running processes in parallel affect the performance of each other additionally to a trivial factor like a workload. In this experiment, we aim at primarily investigating those effects. We run the test on Quad-core 2.9GHz Processor. For simplicity, we consider only three processing kinds of representative time complexities: linear (single for-loop), polynomial (nested for-loop), and exponential (recursive). With a fixed number of the linear process as five, Fig. 8 shows the influence of parallel processes over different load size. The common finding is that the number of processes does affect the processing delay in considering tasks. Furthermore, the different complexity of those processes also incurs different influences as noticeable in Fig. 8a and in Fig. 8b. Despite a small load size, the exponential process adds a larger effect on the considering process comparing to the polynomial process.

5.3 System Deployment

For evaluating our proposed system, we assume the small community environments consisting of four common areas, gym, park, kitchen and office, illustrated in Fig. 9. Given event streams of video frame are available, clients request three recipes applying learning model to recognize activity on gym, park, and kitchen areas from a sampling frame then activate corresponding actions that are adjusting gym air condition (GymActDetect), reporting police (ParkActDetect), and turning and tuning kitchen lights (KitchenActDetect), respectively. In this scenario, the video streams in parks are from dynamic and mobile devices without processor service activated. Therefore, the police-reporting recipe gives more weight to reliability than efficiency and availability. Meanwhile, the light-tuning recipe gets input from a static camera with processing power and requires a low-latency response. Similarly, the air-adjusting recipe processes video streams from a static empowered camera. Conversely, it

recipe	(e,r,a)	$score_h$	$score_m$	$score_l$
GymActivity	(0.4, 0.2, 0.4)	16	9	6
ParkActivity	(0.1, 0.3, 0.6)	12	10	9
KitchenActivity	(0.7, 0.1, 0.2)	20	8	3

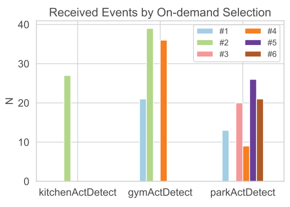
Table 1: QoS weight and selection chance at each performance class of each recipe

#	core	CPU	service	location
1	4	2.9GHz	coordinator,processor,actor	office
2	8	2.6GHz	processor	office
3-4	1	2.9GHz	processor,driver	gym
5	1	2.9GHz	processor,actor	gym
6	1	2.9GHz	processor,driver	kitchen
7-8	2	500MHz	actor	kitchen
9-11	2	500MHz	driver	park
12	2	500MHz	actor	park

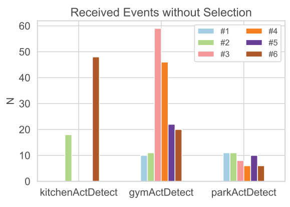
Table 2: CPU spec and installed services and location

class, c	threshold, β_c	guarantee, α_c	eff. advantage, γ_c
High	≥ 40	1.5	0.8
Medium	≥ 20	2	0.2
Low	≥ 0	2.5	0

Table 3: Performance Class Specification



(a) with on-demand selection



(b) without selection

Fig. 10: number of event received at each processing unit

requires higher coverage from multiple cameras. As a result, we set the QoS weight of each recipe, as presented in table 1. For benchmark data sets, we use videos of realistic actions from UFC50 [26] for park and gym activities and first-person vision videos from EPIC Kitchen fir kitchen activity [9]. The processing intervals of each recipe are all set to 0. Hence, all recipes are non-deterministic. Table 2 lists our deployment detail. The first two processors run as a host while the rests execute the program from inside of Docker containers [20]. The deployment on Intel Edison utilizes *BalenaCloud* platform, the container-based platform for deploying applications on IoT devices. The function-operating module works with FlatBuffers file from TensorFlow Lite API [1]. For memory requirement, the device with only actor service requires approximately 500MB, whereas the device with a processor or driver service asks for twice for library dependency. The parameters of performance class, described in section 4, are set as presented in table 3. Each video driver samples and generates a frame event to its postman for every second. The recipe selection performs every minute. We set a batch size for arrival and processing time predictions to 10. The comparing methods are (1) our proposal, (2) proposed selection with round-robin offloading approach, *RRB*, (3) proposed selection without offloading, *No Offload*, and (4) round-robin offloading without task-selection (i.e., place all operators), *No Selection*. The round-robin approaches forcibly en-queue the offloaded input regardless of availability.

According to performance prediction, the processor #1 and #2 are classified as high while the rest (#3 to #6) are medium. Since the #1 runs multiple processes, the highest-spec device is #2. The number of distributed work on each processor is shown in Fig. 10. With the proposed selection, both high-class processors tend to choose the top two recipes when considering weight on efficiency, that is *KitchenActDetect* and *GymActDetect*. *GymActDetect* has two-times replication and distributes over high and medium class. *ParkActDetect* mostly distributes over medium class due to lowest efficiency weight. Without selection, all recipes randomly distributed over all processors depending to network conditions. Seeing that the device number #3 and #4 is the event source of *GymActDetect* and #6 is the event source of *KitchenActDetect*, there has no problem of network traffic compared to the others.

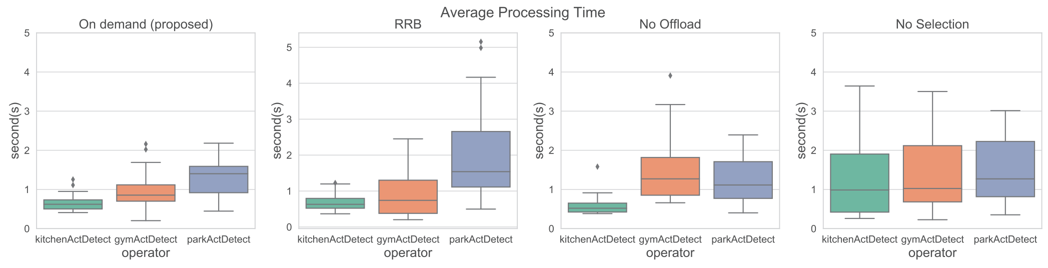
To cut off the network connection bias in comparison results, we use only main-processing time and accumulated time, which includes waiting and offloading time for the evaluation metrics. As presented in Fig. 11, the efficiency-weighted recipe, which runs *KitchenActDetect* operator, achieves low latency from recipe selection decision. The *GymActDetect* and *ParkActDetect* is slightly higher respectively with on-demand offloading method. The RRB approach has noticeably higher processing delay as the high-class devices offload the *GymActDetect* tasks to medium-class devices more frequent with no necessary. Meanwhile, the filterer in No Offload method distributes some of the *GymActDetect* work to the medium-class device and the processing unit never offloads those work to the higher class even if it meets overloading condition. As a result, the processing without offloading suffers from queue waiting. The average time results of all operators in *No Selection* are not significantly different since all devices equally execute all tasks. The different only comes from the number of replications and origin point.

6. Conclusion

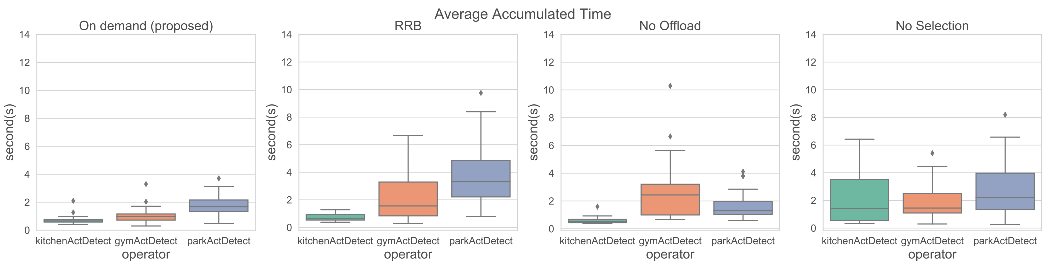
To distribute tasks in a self-organized distributed stream processing system using our developed μ*EdgeCEP* as a paradigm, we introduce two-fold mechanisms for long-term and short-time decision. The former is QoS-oriented task selection. It is a heuristic algorithm to pick up tasks according to the task-specific QoS against processing unit performance under memory and time constraints. The later is on-demand offloading. It applies the queuing theory to determine whether it should offload the task to another neighbor or not. Both mechanisms work independently at each device. The evaluation results highlight the significance of combining both mechanisms from the processing delay metric comparing to the other combination with commonly-found policy.

Acknowledgment

This work was supported by Japan-US Network Opportunity 2 (JUNO2) (No.193)””, the Commissioned Research of National Institute of Information and Communications Technology (NICT), JAPAN



(a) average processing time



(b) average accumulated time

Fig. 11: Delay Comparison Results collected from All Processing Units

References

[1] Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y. and Zheng, X.: TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems (2015).

[2] Arabnejad, H., Barbosa, J. and Suter, F.: Fair Resource Sharing for Dynamic Scheduling of Workflows on Heterogeneous Systems, *High-Performance Computing on Complex Environments* (Jeannot, E. and Zilinskas, J., eds.), Parallel and Distributed Computing series, Wiley (2014).

[3] Awadalla, M. H. A.: Task Mapping and Scheduling in Wireless Sensor Networks, *IAENG International Journal of Computer Science*, Vol. 40, No. 4, pp. 257–265 (2013).

[4] Bosse, S.: Distributed Machine Learning with Self-Organizing Mobile Agents for Earthquake Monitoring, *2016 IEEE 1st International Workshops on Foundations and Applications of Self* Systems (FAS*W)*, pp. 126–132 (2016).

[5] Choochotkaew, S., Yamaguchi, H. and Higashino, T.: A Self-Organized Task Distribution Framework for Module-Based Event Stream Processing, *IEEE Access*, Vol. 7, pp. 6493–6509 (2019).

[6] Choochotkaew, S., Yamaguchi, H., Higashino, T., Schäfer, D., Edinger, J. and Becker, C.: Self-adaptive Resource Allocation for Continuous Task Offloading in Pervasive Computing, *2018 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*, pp. 663–668 (2018).

[7] Cicconetti, C., Conti, M. and Passarella, A.: Low-latency Distributed Computation Offloading for Pervasive Environments, *2019 IEEE International Conference on Pervasive Computing and Communications (PerCom)*, pp. 262–271 (2019).

[8] Cugola, G. and Margara, A.: Processing Flows of Information: From Data Stream to Complex Event Processing, *ACM Comput. Surv.*, Vol. 44, No. 3, pp. 15:1–15:62 (2012).

[9] Damen, D., Doughty, H., Farinella, G. M., Fidler, S., Furnari, A., Kazakos, E., Moltisanti, D., Munro, J., Perrett, T., Price, W. and Wray, M.: Scaling Egocentric Vision: The EPIC-KITCHENS Dataset, *European Conference on Computer Vision (ECCV)* (2018).

[10] Feitelson, D. G.: *Workload Modeling for Computer Systems Performance Evaluation*, Cambridge University Press, New York, NY, USA, 1st edition (2015).

[11] Gallager, R.: *Discrete Stochastic Processes*, The Springer International Series in Engineering and Computer Science, Springer US (2012).

[12] Garcia Lopez, P., Montresor, A., Epema, D., Datta, A., Higashino, T., Iamnitchi, A., Barcellos, M., Felber, P. and Riviere, E.: Edge-centric Computing: Vision and Challenges, *SIGCOMM Comput. Commun. Rev.*, Vol. 45, No. 5, pp. 37–42 (2015).

[13] Heemin Park, J. W. L.: Task Assignment in Wireless Sensor Networks via Task Decomposition, *Information Technology And Control*, Vol. 41, No. 4, pp. 340–348 (2012).

[14] Kounev, S., Nou, R. and Torres, J.: Autonomic QoS-Aware Resource Management in Grid Computing Using Online Performance Models, *Proceedings of the 2Nd International Conference on Performance Evaluation Methodologies and Tools*, ValueTools '07, ICST, Brussels, Belgium, Belgium, ICST, pp. 48:1–48:10 (2007).

[15] Lin, F. C. H. and Keller, R. M.: The Gradient Model Load Balancing Method, *IEEE Transactions on Software Engineering*, Vol. SE-13, No. 1, pp. 32–38 (1987).

[16] Lo, V. M.: Heuristic algorithms for task assignment in distributed systems, *IEEE Transactions on Computers*, Vol. 37, No. 11, pp. 1384–1397 (1988).

[17] Mao, J., Jannotti, J., Akdere, M. and Cetintemel, U.: Event-based Constraints for Sensornet Programming, *Proceedings of the Second International Conference on Distributed Event-based Systems*, DEBS '08, New York, NY, USA, ACM, pp. 103–113 (2008).

[18] Marcmarais: A Quad Core at 2.66Ghz or a Dual Core at 3.33GHz?

[19] Mayer, R., Koldehofe, B. and Rothermel, K.: Predictable Low-Latency Event Detection With Parallel Complex Event Processing, *IEEE Internet of Things Journal*, Vol. 2, No. 4, pp. 274–286 (2015).

[20] Merkel, D.: Docker: Lightweight Linux Containers for Consistent Development and Deployment, *Linux J.*, Vol. 2014, No. 239 (2014).

[21] Nakamura, Y., Mizumoto, T., Suwa, H., Arakawa, Y., Yamaguchi, H. and Yasumoto, K.: In-Situ Resource Provisioning with Adaptive Scale-out for Regional IoT Services, *The Third ACM/IEEE Symposium on Edge Computing (SEC 2018)* (2018).

[22] Narkhede, N., Shapira, G. and Palino, T.: *Kafka: The Definitive Guide Real-Time Data and Stream Processing at Scale*, O'Reilly Media, Inc., 1st edition (2017).

[23] Ottenwälder, B., Koldehofe, B., Rothermel, K., Hong, K. and Ramachandran, U.: RECEP: selection-based reuse for distributed complex event processing, *DEBS* (2014).

[24] Park, H. and Lee, J.: Task assignment and migration inwireless sensor networks via task decomposition, *Information Technology and Control*, Vol. 41, pp. 340–348 (2012).

[25] Pietzuch, P., Ledlie, J., Shneidman, J., Roussopoulos, M., Welsh, M. and Seltzer, M.: Network-Aware Operator Placement for Stream-Processing Systems, *Proceedings of the 22Nd International Conference on Data Engineering*, ICDE '06, Washington, DC, USA, IEEE Computer Society, pp. 49– (2006).

[26] Reddy, K. K. and Shah, M.: Recognizing 50 Human Action Categories of Web Videos, *Mach. Vision Appl.*, Vol. 24, No. 5, pp. 971–981 (2013).

[27] Shi, W., Cao, J., Zhang, Q., Li, Y. and Xu, L.: Edge Computing: Vision and Challenges, *IEEE Internet of Things Journal*, Vol. 3, No. 5, pp. 637–646 (2016).

[28] You, K., Tang, B., Qian, Z., Lu, S. and Chen, D.: QoS-aware Placement of Stream Processing Service, *J. Supercomput.*, Vol. 64, No. 3, pp. 919–941 (2013).