Dynamic Tuple Scheduling with Prediction for Data Stream Processing Systems

Xi Huang, Ziyu Shao, Yang Yang
School of Information Science and Technology, ShanghaiTech University
Email: {huangxi, shaozy, yangyang}@shanghaitech.edu.cn

***Abstract*—For data stream processing systems such as Apache Heron, workload imbalance across processing instances often causes significant system performance degradation. To mitigate such issues, Apache Heron leverages a naive throttling-based back-pressure scheme, which may lead to unexpected system disruption. This calls for a finer-grained control to distribute data stream units (tuples) between successive instances, *a.k.a. tuple scheduling*, which well adapts to data stream variations and workload discrepancy. Besides, the benefits of predictive scheduling to data stream processing systems still remain unexplored. In this paper, we formulate tuple scheduling problem as a stochastic network optimization problem, with careful choices in the granularity of system modeling and decision making. With non-trivial transformation, we decouple the problem into a series of online subproblems. By exploiting unique subproblem structure, we propose *POTUS*, an efficient, online, and distributed scheduling scheme that employs the power of predictive scheduling but requires only limited system dynamics to achieve a tunable trade-off between communication cost reduction and system queue stability. Theoretical analysis and simulations show that POTUS effectively shortens response time with mild-value of future information, even in face of mis-prediction. Our solution is also applicable to other data stream processing systems.**

I. Introduction

Modern online service providers maintain their data stream processing systems to conduct large-scale analytic services in real time [1]–[7]. The most updated systems, *e.g.*, Heron [8], are designed with particular aim at high scalability, availability, and fault tolerance. It has been widely adopted by numerous organizations such as Google, Twitter, Alibaba, and Facebook.

In Heron, each data stream processing application is built as a *directed-acyclic graph* (DAG), *a.k.a.* a *topology*, where data streams (in the unit of *tuples*) are processed in pipeline through a series of components [8]. The instances of components are run as independent processes and packed into containers to ensure performance isolation. Due to high cost of migration and instantiation, instance re-placement is usually performed infrequently and thus can be considered fixed at runtime for tuple-scale operations. With a fixed instance placement, instances' output tuples are forwarded to their successive instances across containers, *a.k.a. tuple scheduling*.

Conducting effective tuple scheduling is non-trivial, which often comes down to the uncertainties in tuple traffic dynamics. At runtime, the amounts of tuple traffic often vary over a wide range in the temporal dimension [9]. Such variations, if not coped with properly, can cause instances being overloaded or even fatal system breakdown, thereby inducing undesirably long response time to real-time applications [10].

To address such issues, Heron, if not specified, distributes tuples uniformly at random from instances to their successors, and employs a naive back-pressure scheme that throttles all ingress components (*a.k.a. spouts*) when any instance is found overloaded. Such a scheme, though easy to implement and responsive to traffic change, also brings about disadvantages. On one hand, it implicitly assumes the homogeneity of instances' processing capacity, but often times this is not the case in practice [11] and the difference between processing capacities can still induce imbalanced workloads and overloaded instances. On the other hand, throttling-based back-pressure can severely hurt application performance, inducing tuple loss and hence degraded quality-of-service.

So far, it is still an open problem to design an adaptive, online, and efficient tuple scheduling scheme that achieves balanced workloads across instances, so that one trades little overheads for significant improvement in tuple response time.

A further question to such a design is: if future tuple arrivals can be predicted, even within a short time window, what is the *fundamental* benefits of such information to tuple scheduling? Recently, learning-aided predictive scheduling has been adopted in many systems; *e.g.*, Netflix prefetches videos of interest onto user-end devices based on user-behavior prediction [12]. However, such pre-service, if wrongly decided, may bring adverse effects. To stream processing systems, the benefits of predictive scheduling and the impact of mis-prediction still remains unknown. A systematic study is essential to the system designers to understand the endeavor worthy to put on predictive scheduling.

In this paper, we aim to address the above challenges. Our key results and contributions are summarized as follows.

System Modeling and Formulation: We formulate the tuple scheduling problem as a stochastic network optimization problem, based on a dedicatedly developed model which captures the interplay between instances at the tuple level and conducts decision making on a time-slot basis to avoid the considerable overheads brought by per-tuple optimization.

Algorithm Design: We propose *POTUS*, a Predictive Online Tuple Scheduling scheme that schedules tuples between successive instances in a distributed manner. Our performance analysis shows that POTUS achieves a near-optimal communication cost while stabilizing system queue backlogs.

Predictive Scheduling: To our best knowledge, this paper is the first to integrate predictive scheduling into data stream processing systems with a systematic study on the benefits of predictive scheduling therein. Our solution can be extended to other data stream processing engines.

Experiment Verification and Investigation: By adapting existing task placement schemes to Heron, our simulation results show that POTUS outperforms such baseline schemes with close-to-optimal communication cost and ultra-low response time. In addition, only mild-value of future information suffices to significantly shorten tuple response time, even in face of mis-prediction.

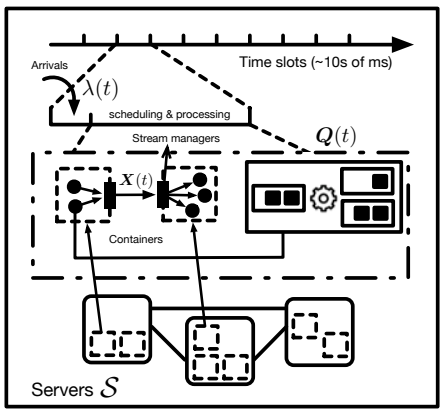


Fig. 1. An illustration of the overall system model. The system proceeds over time slots and consists of a set of servers S, each hosting some containers. For data stream processing applications, their components' instances (solid circles) are packed in containers with fixed placement. Each instance maintains input/output queues, denoted by vector $\boldsymbol{Q}(t)$, to buffer tuples (solid squares in queues). In each time slot t, instances receive a number $\lambda(t)$ of new tuples, process some of them, and output some of them to output queues. Then the stream manager in each container makes the tuple scheduling decision $\boldsymbol{X}(t)$ and forward tuples between successive instances.

The rest of this paper is organized as follows. Section II presents our model and formulation. Section III shows the design of POTUS, followed by its theoretical analysis. Section IV shows simulation results and analysis and Section V concludes the paper.

II. Problem Formulation

We consider a Heron-based data stream processing system that hosts various applications in a shared cluster infrastructure, with fixed placement of instances determined by some particular schemes [10]. The system proceeds over time slots $t \in \{0, 1, 2 \dots\}$. Each slot has a length of tens of milliseconds, proportional to the average tuple processing latency [6]. In every time slot, instances process and forward tuples to subsequent instances through stream managers. An illustration of our system model is shown in Figure 1.

A. Streaming Application Model

A set of stream processing applications are running in the system, denoted by $\mathcal{A}$. Each application is represented as a directed acyclic graph (DAG), with processing components as nodes and data streams as edges. We denote by $\mathcal{C}$ the set of all components in applications $\mathcal{A}$. In practice, the diameter of each application's DAG is usually not very large, mostly no more than three [6].

B. Deployment Model

All applications are deployed in a cluster of heterogeneous servers, denoted by set $\mathcal{S}$. At runtime, each component runs as multiple instances for scalability. We denote by $\mathcal{I}$ the set of all instances in the system. For instance $i \in \mathcal{I}$, between any successive instances i and i', the average tuple traffic rate is denoted by $\bar{R}_{i,i'}$, which is often attainable from the system. To distinguish instances by their components, we define $\mathcal{I}_{\mathrm{C}}(c)$ as the set of instances for component $c \in \mathcal{C}$. Further, for each instance $i \in \mathcal{I}$, we denote the set of its preceding components by $p(i)$ ($\emptyset$ for spout instances), and the set of its subsequent components by $n(i)$ ($\emptyset$ for terminal bolt instances).

Instances run as independent processes and are packed into containers, while containers reside on servers. We denote the set of containers by $\mathcal{K}$. For each container $k \in \mathcal{K}$, we denote the set of instances that it hosts by $\mathcal{I}_K(k)$.

C. Decision Variables for Tuple Scheduling

Once deployed, in every time slot t, each instance i makes a set of tuple scheduling decisions $X_{i,*}(t)$, such that $X_{i,i'}(t)$ decides the number of tuples to send to instance i' for $c' \in n(i)$ and $i' \in \mathcal{I}_C(c')$, *i.e.*, one of the instances of some subsequent component of instance i. If i and i' are not successive instances, then $X_{i,i'}(t) = 0$. Besides, each instance i can transfer at most γ_i tuples, *i.e.*,

$$\sum_{c' \in n(i)} \sum_{i' \in \mathcal{C}(c')} X_{i,i'}(t) \leq \gamma_i. \tag{1}$$

D. Queueing Model

In Heron, instances often maintain queue backlogs to buffer tuples [8]. Particularly, for any instance i, it buffers new tuples in an output queue. Such new tuples are either generated from data streams (if i is a spout instance), or produced by previous processing (if i is a bolt instance). Depending on the processing logic, these tuples may be sent to instances of more than one bolt. We distinguish tuples with different targeted bolts by introducing a virtual output queue for successive bolts $n(i)$. In time slot t, each $c' \in n(i)$ possesses a backlog of size $Q^{(\mathrm{out})}_{i,c'}(t)$. Such a multi-queue model is logically equivalent to the physical output queue, while accurately characterizing the tuple forwarding process on individual instances. Note that we omit the output queue for any terminal bolt instance i.

Meanwhile, if instance i is a bolt instance, then besides the output queue for new tuples, it also maintains an input queue that buffer tuples sent from preceding instances, denoted by $Q^{(\mathrm{in})}_i(t)$. For notational simplicity, we denote all queue backlog sizes at time t by $\boldsymbol{Q}(t)$. Next, we elaborate the queueing dynamics in detail.

Queueing with Predictive Tuple Arrivals: To investigate the *fundamental* benefits of predictive scheduling, we consider a system which can perfectly predict and per-serve future tuple arrivals in a limited lookahead window.[1] We do not assume any particular prediction techniques; instead, we consider the prediction as output from other standalone predictive modules [13]. Accordingly, tuples will be pre-admitted by spout instances and then pre-allocated with resources to shorten their response time.

Under such settings, the system workflow proceeds as follows. At the beginning of each time slot t, based on the instant system dynamics, instances make their scheduling decisions $\boldsymbol{X}(t)$. In particular, spout instances emit new tuples, including those pre-admitted into the systems; in the meantime, both spout and bolt instances forward tuples from their output queues to their succeeding instances. Next, each bolt instance processes tuples from its input queue and generate new ones to output queues. We show the queueing model in Figure 2.

In the following, we specify the queueing dynamics in each time slot t on spout and bolt instances, respectively.

[1] We evaluate the impact of mis-prediction in the simulations.

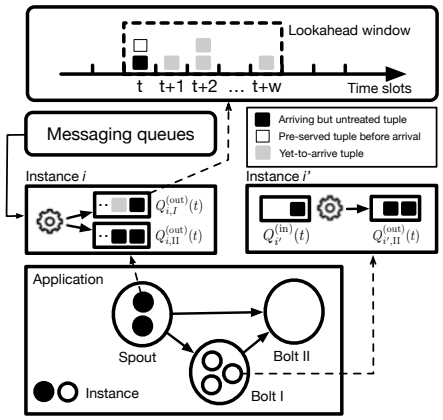


Fig. 2. An illustration of the queueing model.

Queue Backlogs on Spout Instances: On spout instance i, there are $\lambda_{i,c'}(t)$ ($\leq \lambda_{\max}$ for some constant $\lambda_{\max}$) new tuples arriving into its output queue backlog $Q^{(\text{out})}_{i,c'}(t)$. Tuple arrivals are assumed independent over time slots and among different instances. Then for spout instance i, it is assumed to have full access to the future tuples in a lookahead window of size W_i ($\leq W_{\max}$ for some constant $W_{\max}$) before their actual arrivals, denoted by $\{\lambda_{i,c'}(t+1), \ldots, \lambda_{i,c'}(t+W_i)\}$ for $c' \in n(i)$. We denote by $Q^{(\text{rem})}_{i,c'}(t,w)$ ($0 \leq w \leq W_i$) the number of untreated tuples for time $(t+w)$ by time slot t. With pre-service, we have

$$0 \leq Q^{(\text{rem})}_{i,c'}(t,w) \leq \lambda_{i,c'}(t+w), \tag{2}$$

because some tuples may have been pre-admitted or even pre-served in previous time slots. Note that $Q^{(\text{rem})}_{i,c'}(t,0)$ is the number of untreated tuples that actually arrive by time t. Hence, in time t, the output queue for bolt $c' \in n(i)$ actually buffers all untreated tuples in the next W_i time slots, with a total backlog size of

$$Q^{(\text{out})}_{i,c'}(t) = \sum_{w=0}^{W_i} Q^{(\text{rem})}_{i,c'}(t,w). \tag{3}$$

We assume that with scheduling decision $\boldsymbol{X}(t)$, tuples are routed in a fully efficient manner [13]. Consequently, a number of $\delta_{i,c'}(t,w)$ tuples will be forwarded from queue backlog $Q^{(\text{rem})}_{i,c'}(t,w)$. By defining $[x]^+ \triangleq \max\{x,0\}$, we have the following update equations for $Q^{(\text{rem})}_{i,c'}(t,w)$.

- For $0 \leq w \leq W_i - 1$,

$$Q^{(\text{rem})}_{i,c'}(t+1,w) = \left[Q^{(\text{rem})}_{i,c'}(t,w+1) - \delta_{i,c'}(t,w+1)\right]^+. \tag{4}$$

- For $w = W_i$,

$$Q^{(\text{rem})}_{i,c'}(t+1,W_i) = \lambda_{i,c'}(t+W_i+1). \tag{5}$$

Therefore, in time slot t, $Q^{(\text{out})}_{i,c'}(t)$ evolves as follows,

$$Q^{(\text{out})}_{i,c'}(t+1) = \left[Q^{(\text{out})}_{i,c'}(t) - \sum_{i' \in \mathcal{I}_C(c')} X_{i,i'}(t)\right]^+ + \lambda_{i,\eta}(t+W_i+1). \tag{6}$$

Queues Backlogs on Bolt Instances: In each time slot t, instance i has a processing capacity of $\mu_i(t)$ ($\leq \mu_{\max}$ for some constant $\mu_{\max}$) and an input queue backlog of size $Q^{(\text{in})}_i(t)$. After processing, instance i will generate $\nu_{i,c'}(t)$ tuples to its output queue $Q^{(\text{out})}_{i,c'}(t)$, $\forall\, c' \in n(i)$. All such information is attainable from system modules such as metric managers and stream managers at runtime. With scheduling decision $\boldsymbol{X}(t)$, $Q^{(\text{in})}_i(t)$ and $Q^{(\text{out})}_{i,c'}(t)$ are updated as follows.

For input queue backlog $Q^{(\text{in})}_i(t)$, it evolves as

$$Q^{(\text{in})}_i(t+1) = \left[Q^{(\text{in})}_i(t) + \sum_{c' \in p(i)} \sum_{i' \in \mathcal{I}_C(c')} X_{i',i}(t-1) - \mu_i(t)\right]^+. \tag{7}$$

For output queue backlog $Q^{(\text{out})}_{i,c'}(t)$, it is defined for instances of non-terminal bolt instances, and updated as follows

$$Q^{(\text{out})}_{i,c'}(t+1) = \left[Q^{(\text{out})}_{i,c'}(t) - \sum_{i' \in \mathcal{I}_C(c')} X_{i,i'}(t)\right]^+ + \nu_{i,c'}(t). \tag{8}$$

Regarding all spout and bolt instances, to maximize the utilization of transmission capacity, we have

$$\sum_{i' \in \mathcal{I}_C(c')} X_{i,i'}(t) \leq Q^{(\text{out})}_{i,c'}(t), \tag{9}$$

for any instance $i \in \mathcal{I}$ and component $c' \in n(i)$.

E. Optimization Objectives:

To shorten tuple response times, it is essential to minimize the total *communication cost*, often in terms of round-trip time or number of hops. In time slot t, we denote by $U_{k,k'}(t)$ the unit communication cost of sending a tuple from container k to k', and the set of all such costs by $\boldsymbol{U}(t)$.

Given decision $\boldsymbol{X}(t)$, the total communication cost is

$$\Theta(t) \triangleq \hat{\Theta}(\boldsymbol{X}(t)) = \sum_{i,\, i' \in \mathcal{I}} X_{i,i'}(t) U_{k(i),k(i')}(t), \tag{10}$$

where $k(i)$ and $k(i')$ denote the containers that host instance i and i', respectively.

Queueing Stability: Reducing the queueing delay also conduces to shortening tuple response time. By *Little's Theorem* [14], the queue backlog size is proportional to the average response time. Besides, it's also been verified that overloading any instance cancels the benefit from reducing communication costs [10]. Therefore, the other optimization objective is to forward as many tuples as possible while balancing the queue backlogs on different instances. We denote the weighted total queue backlog size in time slot t as

$$h(t) \triangleq \hat{h}(\boldsymbol{Q}(t)) = \sum_{i \in \mathcal{I}} Q^{(\text{in})}_i(t) + \beta \sum_{i \in \mathcal{I}} \sum_{c' \in n(i)} Q^{(\text{out})}_{i,c'}(t), \tag{11}$$

where β is a positive constant that weights the importance of balancing the output queues compared to input queues on instances, respectively. In practice, one can set the value of parameter β proportional to the ratio between the capacity on

the input queue backlogs and the output queue backlogs of instances. Accordingly, we define queueing stability [15] as

$$\limsup_{T\to\infty}\frac{1}{T}\sum_{t=0}^{T-1}\mathbb{E}\left\{h(t)\right\}<\infty. \tag{12}$$

F. Problem Formulation

We formulate the following *tuple scheduling problem* which aims to minimize time-average communication cost in the long run.

$$\begin{aligned}\underset{\{\boldsymbol{X}(t)\}_t}{\text{Minimize}}\quad & \limsup_{T\to\infty}\frac{1}{T}\sum_{t=0}^{T-1}\mathbb{E}\left\{\hat{\Theta}\left(\boldsymbol{X}(t)\right)\right\}\\ \text{Subject to}\quad & (1),(9),(12).\end{aligned} \tag{13}$$

III. Algorithm Design and Performance Analysis

A. Algorithm Design

To solve problem (13), we adopt Lyapunov optimization techniques [15] to transform the long-term stochastic network optimization problem into a series of subproblems to be solved over time slots.[2] In particular, in time slot t, we aim to solve the following problem:

$$\begin{aligned}\underset{\boldsymbol{X}}{\text{Minimize}}\quad & \sum_{i\in\mathcal{I}}\sum_{c'\in n(i)}\sum_{i'\in\mathcal{I}_C(c')}l_{i,i'}(t)X_{i,i'}\\ \text{Subject to}\quad & \sum_{c'\in n(i)}\sum_{i'\in\mathcal{C}(c')}X_{i,i'}(t)\le\gamma_i,\ \forall\, i\in\mathcal{I},\\ & \sum_{i'\in\mathcal{I}_C(c')}X_{i,i'}\le Q^{(\text{out})}_{i,c'}(t)\ \forall\ i \text{ and } c'\in n(i),\end{aligned} \tag{14}$$

where we define

$$l_{i,i'}(t)\triangleq V\cdot U_{c(i),c(i')}(t)+Q^{(\text{in})}_{i'}(t)-\beta Q^{(\text{out})}_{i}(t) \tag{15}$$

as a positive constant given $\boldsymbol{U}(t)$, $\boldsymbol{Q}(t)$, and positive parameter V at the beginning of time slot t. In fact, problem (14) can be decomposed on a per-instance basis and every instance makes its tuple scheduling decision independently. Nonetheless, it is not practical for implementation, because per-instance optimization requires extra resource from each instance to keep its related states and undertake decision making, inducing considerable overheads. Instead, the decision making for tuple scheduling can be handled by the stream manager in each container, since:

(i) Stream manager itself manages the control of tuple transmission between instances within and across containers; hence it naturally keeps the instant information required to calculate (15) such as $\boldsymbol{U}(t)$.
(ii) The queue backlog sizes $\boldsymbol{Q}(t)$ of all its instances can be attained by direct interaction with the module that holds such information in each container, *i.e.*, metric manager.
(iii) The number of instances in every container is often not very large, about ten on average [16].

The above discussion suggests that it is a proper choice to conduct tuple scheduling within stream managers, which also minimizes the impact of decision making process on instances' execution. In the following, we propose POTUS, an efficient and distributed scheme that solves problem (14) optimally and present its pseudocode in Algorithm 1.

[2]We refer all detailed derivation to Appendix-A.

Algorithm 1 POTUS (Predictive Online Tuple Scheduling) in one time slot

1: At the beginning of time slot t, the stream manager of each container k collects instant system dynamics: $\boldsymbol{Q}(t)$ from its metric manager, and estimates $U_{k,*}(t)$.
2: **For** each instance $i\in\mathcal{I}_K(k)$
3: Initialize the number of tuples to be sent in time slot t as $\tilde{\gamma}_i(t)=0$.
4: Pick out candidate instances $\mathcal{I}_{\text{cand}}(i)$ such that
$$\mathcal{I}_{\text{cand}}(i)\leftarrow\left\{i'\in\mathcal{I}_C(c'),\forall\, c'\in n(i)|l_{i,i'}(t)<0\right\}.$$
5: **If** $\mathcal{I}_{(\text{cand})}(i)=\emptyset$ then
6: Set $X_{i,i'}(t)=0$ for all $i'\in\mathcal{I}_C(c'),\forall\, c'\in n(i)$.
7: **For** each subsequent component $c'\in n(i)$
8: Initialize $\tilde{Q}^{(\text{out})}_{i,c'}(t)\leftarrow Q^{(\text{out})}_{i,c'}(t)$.
%% Making decisions for tuple scheduling
9: **While** $\tilde{\gamma}_i(t)<\gamma_i$ and $\mathcal{I}_{\text{cand}}(i)\neq\emptyset$:
10: Pick such instance i^* of component c^* that
$$i^*\in\underset{i'\in\mathcal{I}_{\text{cand}}(i)}{\arg\min}\ l_{i,i'}(t).$$
11: Set $X_{i,i^*}(t)\leftarrow\min\{\gamma_i-\tilde{\gamma}_i(t),\tilde{Q}^{(\text{out})}_{i,c^*}(t)\}$.
12: Update $\tilde{\gamma}_i(t)\leftarrow\tilde{\gamma}_i(t)+X_{i,i^*}(t)$.
13: Update $\tilde{Q}^{(\text{out})}_{i,c^*}(t)\leftarrow\max\{\tilde{Q}^{(\text{out})}_{i,c^*}(t)-X_{i,i^*}(t),0\}$.
14: Update $\mathcal{I}_{\text{cand}}(i)\leftarrow\mathcal{I}_{\text{cand}}(i)\backslash\{i^*\}$.
15: Update instances' queue backlogs according to (6) – (8).

Remark: In particular, POTUS conducts the scheduling decisions in a distributed manner on containers. By collecting all necessary information, the stream manager in each container can make scheduling decisions independently for its instances. The calculation of (15) only requires the communication cost to other containers, its output queue backlog size, and the input queue backlog size on the target instance. Particularly, for instance i, $l_{i,i'}(t)$ actually reflects the unit price of transferring a tuple from instance i to instance i'. When an instance is ready to forward its tuples, it would avoid heavily loaded successors, and in turn chooses relatively less loaded ones. Meanwhile, POTUS also takes communication cost into consideration, by favoring the instances successors which stay in containers close to its current container. The final scheduling decision is reached by evaluating all the above factors. In such a way, POTUS can distribute tuple traffic adaptively to avoid overloading any individual instance while incurring low communication costs.

B. Performance Analysis for POTUS

According to Algorithm 1, we analyze the time complexity of POTUS in one time slot. First, for every instance i, POTUS requires $O(C_{\max}\cdot I^C_{\max})$ time to form a candidate set from all its succeeding instances (line 4), where $C_{\max}$ denotes the maximum number of components in any application and $I^C_{\max}$ denotes the maximum number of instances (parallelism) of any component. If the candidate set is not empty, then the tuple scheduling loop (line 12-18) takes at most $C_{\max}\cdot I^C_{\max}$ iterations in the worst case, corresponding to the case with all successors of instance i being scheduled in the process. During each iteration, picking the target instance (line 13) also requires at most $O(C_{\max}\cdot I^C_{\max})$ iterations to finish.

Therefore, the time complexity for tuple scheduling loop is $O\left(\left(C_{\max} \cdot I_{\max}^{C}\right)^2\right)$. Consequently, the computational complexity of POTUS for each instance i is $O((C_{\max} \cdot I_{\max}^{C})^2)$. To analyze the overall complexity, remind that the stream manager makes scheduling decisions for its residing instances, independent of the others. Hence, with necessary information attained, stream managers can undertake the scheduling in parallel, with an overall complexity of $O(I_{\max}^{K} \cdot (C_{\max} \cdot I_{\max}^{C})^2)$, where $I_{\max}^{K}$ denotes the maximum number of instances in any container. In practice, the number of components and the number of instances per container are usually no very large [6] [16], and thus POTUS trades off only little overheads for scheduling tuples with fine-grained control.

On the other hand, when all lookahead window sizes are zero ($W_i = 0$ for all i), *i.e.*, no future information is available, POTUS degenerates to the classical Lyapunov optimization algorithm and achieves an $[O(V), O(1/V)]$ trade-off between the time-average total queue backlog size and the time-average total communication cost via a tunable parameter V, while guaranteeing the stability of queue backlogs in the system. We refer the proof to Appendix-B. When $W_i \neq 0$, *i.e.*, future information is available, POTUS conducts predictive scheduling and achieves much better performance, *e.g.*, shorter response time, as will be shown in the simulations.

IV. Simulation and Evaluation

A. Simulation Settings

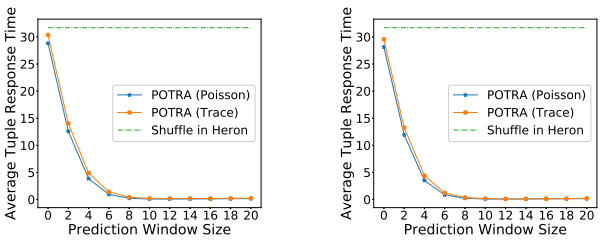
We construct two stream processing systems based on two widely adopted topologies, Jellyfish [17] and Fat-Tree [18]. Each of them contains 24 switches and 16 servers that host stream processing applications.

Stream Processing Applications: In the simulation, we deploy five data stream processing applications with common topologies [7] [10] [19]. Each application has a topology depth varying from 3 to 5 and a number of components ranging from 3 to 6. Instances of the same component have the identical processing capacity ranging from 3-5 tuples per time slot.

Instance Placement: For instance placement, the mapping from instances to containers is determined by T-Heron, a placement scheme which is adapted from T-Storm [10]. Given a new application, T-Heron sorts all its instances by their descending order of (incoming and outgoing) tuple traffic rate. Then it iteratively assigns each instance to one of the available containers with minimum incremental traffic.

Traffic Workloads: We conduct trace-driven simulations with tuple arrival measurements drawn from real-world network systems [20]. In addition, we also conduct simulations where tuple arrivals follow Poisson distribution, with the same arrival rate as in trace-driven cases.

Prediction Settings: The traffic of different applications often varies in predictability. By fixing the average window size as W, the prediction window size for each application is set by sampling uniformly from $[0, 2 \times W]$ at random. We evaluate cases with perfect and imperfect prediction. For *perfect prediction*, future tuple arrivals in the time window are assumed perfectly predicted and can be pre-served. For *imperfect prediction*, we implement five forecasting schemes that predict tuples yet to arrive (all with $W = 1$), with mean-square error (MSE) varying from 10.37 to 22.54, including: 1) *Kalman filter* [21]; 2) distribution estimator (*Distr*), which predicts the number of arriving tuples in each future time slot by independent sampling from the empirical distribution of the frequency of arriving tuple numbers in the past time slots; 3) *Prophet* [22], Facebook's time-series forecasting procedure; 4) moving average (*MA*) and 5) exponentially weighted moving average (*EWMA*) [23]. Then we consider two basic types of mis-prediction. One is *true negative*, *i.e.*, when an actual tuple is not predicted to arrive and hence not pre-served before its arrival. The other is *false positive*, *i.e.*, when a tuple that does not exist is predicted to arrive; in this case, processing such tuples consumes extra system resources. To investigate the limits of predictive scheduling, we compare predictive scheduling with perfect prediction against two extremes of the spectrum: 1) all actual tuple arrivals fail to be predicted; 2) the actual arrivals are correctly predicted, but with some extra false-positive arrivals.



(a) Jellyfish Topology (b) Fat-Tree Topology

Fig. 3. Average response time vs. look ahead window size with Poisson and trace arrival process, under different topologies.

Compared Baselines for Tuple Scheduling: All instances are assumed stateless so that for any component, any of its instances can provide identical processing to the incoming tuples. We compare POTUS with shuffling, *i.e.*, the default tuple scheduling scheme in Heron, which dispatches each tuple to one of a component's instances uniformly at random.

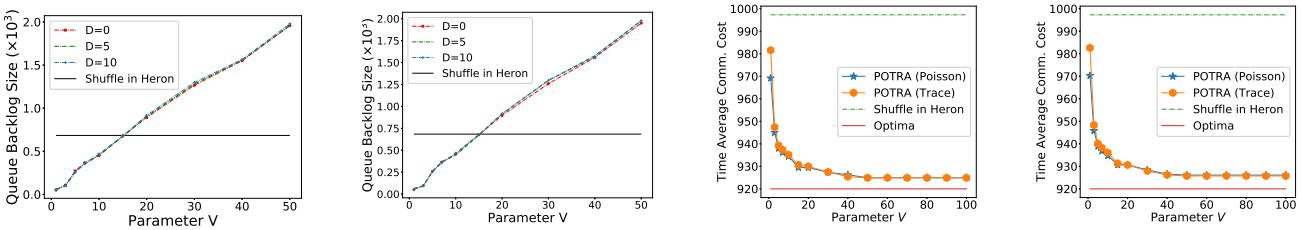
Metric of Response Time: For each tuple, its response time is counted as the number of time slots from its actual arrival to the last completion of its descendant tuples. If a tuple is pre-served before its actual arrival, then it will be responded instantly once it arrives, with a zero response time.

B. Performance Evaluation and Analysis

We fix the instance placement and evaluate the performance of POTUS under both perfect and imperfect predictions.

1) Performance Evaluation under Perfect Prediction: To investigate the benefits of predictive scheduling, we compare POTUS with zero ($W = 0$) and non-zero lookahead window sizes under different settings. The former is a special case of POTUS without future information.

Average response time vs. lookahead window size W: Figure 3 (a) and (b) show the results under Jellyfish and Fat-Tree topology, respectively. We take the curve of Jellyfish under Trace arrival as an example: as window size increases from 0 to 6, POTUS shortens the average response time from $31.4ms$ to $1.5ms$. As W continues increasing, the marginal reduction in average response time diminishes. Eventually, the response time remains stably at $0.5ms$. Likewise, curves under Fat-Tree topology also exhibit similar trends. On the other hand, shuffle scheme incurs an average response time about 5% higher than POTUS. This is reasonable since shuffle implicitly assumes the homogeneity of instances' processing capacities while POTUS exploits instant system dynamics.



(a) Jellyfish Topology (b) Fat-Tree Topology (c) POTUS ($W = 0$) (d) POTUS ($W = 10$)

Fig. 4. Performance of POTUS under various settings with perfect prediction.

Insight: Results from Figure 3 highlight the benefits of predictive scheduling in shortening the average tuple response time. Moreover, mild-value of future information is sufficient to effectively reduce tuple response time down to nearly zero. The marginal benefits then diminish with more future information due to the limit in system service capacity.

Time-average total queue backlog size: Figure 4 (a) and (b) show how the total system queue backlog changes under different topologies, window sizes, and the values of parameter V. From both plots, we observe that: 1) the shuffle scheme incurs a constant backlog size since its has nothing to do with the value of V; 2) with different window sizes, POTUS incurs a linearly ascending trend in the queue backlog size as the value of V grows from 1 to 50. Particularly, under Fat-Tree topology, as the value of V becomes greater than 16, the system cumulates more queue backlog by POTUS than the shuffle scheme in Heron. This is because a larger value of parameter V leads instances to forward more tuples to those with low communication cost among their successive instances, resulting in more unbalanced workloads across instances. In contrast, with a small value of V, POTUS focuses on steering tuples between instances to stabilize queue backlogs and induce more balanced workloads. This implies that, in practice, one should greedily choose a small value of parameter V. However, as we shall see later, the choice of V also decides how the communication cost changes and thereby a trade-off must be made.

Considering that the results under different topologies and arrivals processes are similar, in the following, we focus on the results under Fat-Tree topology and trace-driven arrivals.

Time-average communication cost vs. V: Figure 4 (c) and (d) evaluate the performance of POTUS in terms of time-average total communication cost as the value of parameter V varies from 1 to 100, with $W = 0$ and $W = 10$, respectively.

Figure 4 (c) shows that POTUS outperforms the shuffle scheme in Heron by up to 7.6% reduction in the communication cost. Besides, for the curves of POTUS, we also see the dramatic and rapid descending trend of communication cost by increasing the value of V from 1 to 100. Particularly, when the value varies from 1 to 40, the time-average communication cost decreases by 5.8%; however, if further increasing V, we see the cost reduction stops and remains constant after $V = 50$, leaving a 0.5% gap to the optimal cost. In fact, the optimality gap is the price to take for stabilizing the system queue backlogs. To keep queue backlog stability, they would not choose their proximal instances that are heavily loaded. Instead, they turn to other less loaded instances with larger communication costs, thereby inducing a sub-optimal total cost. The descending trend remains the same for both $W = 0$ and 5. This shows that under perfect prediction, predictive scheduling incurs almost no extra system cost.

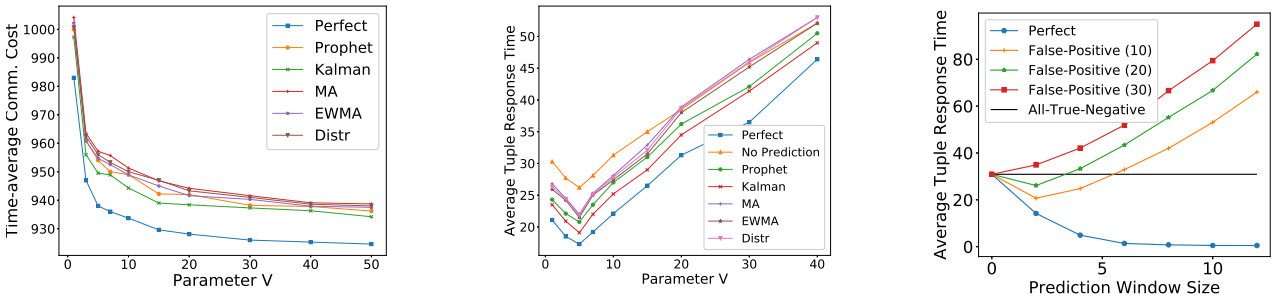
Insight: Figure 4 (a) - (d) show that, by increasing the value of parameter V, not only does the queue backlog size increase, but also the communication cost is reduced. Therefore, parameter V actually determines the trade-off between communication cost and queue backlog size in the system, as shown by our previous analysis in Section III-B. By choosing a relatively small value for parameter V, *e.g.*, $V = 5 \sim 15$ in our simulation, we can attain both lower communication cost and smaller queue backlog size. In practice, the value can be tuned according to particular objectives in system design.

2) Performance Evaluation under Imperfect Prediction: Prediction errors are inevitable in practice. In this subsection, we first evaluate the performance of POTUS given perfect prediction and five forecasting schemes, including *Kalman*, *Distr*, *Prophet*, *MA*, and *EWMA*, which may incur mis-prediction such as false positive and true negative of tuple arrivals. For these schemes, we set the window size as $W = 1$, considering that highly varying traffic dynamics and statefulness of tuples often confine systems to conduct short-term forecasting on the future arrivals. We also investigate the fundamental benefits of predictive scheduling in face of mis-prediction.

Time-average communication cost vs. V: Figure 5 (a) shows the time-average communication cost induced by POTUS with perfect prediction and five forecasting schemes. We see a rapid fall of communication cost for all schemes. With perfect prediction, the communication cost is even lower than the other five forecasting schemes. The superfluous cost is caused by mis-prediction, especially false-alarm, which misleads the system to allocate extra resources to process tuples that does not exist. Therefore, improving forecasting accuracy conduces to reducing communication cost.

Average response time vs. V: Figure 5 (b) tells about how the tuple response time varies under different choices of parameter V. We make the following observations.

First, as V increases from 0 to 5, we first a reduction in average response time by about 20% and then a linear growth of response time thereafter. Such a trend change in response time is mainly determined by two factors, tuples' queueing delay and the communication cost. We know that increasing the value of parameter V leads to the reduction in communication cost and growth in queue backlog size. Remind that a greater queue backlog size, by *Little's theorem* [14], implies longer queueing delay for tuples. From Figure 5 (a), we see a significant reduction in the communication cost as the value of parameter V rises from 1 to 10. Therefore, in



(a) Communication Cost with Parameter V (b) Tuple Response Time with Parameter V (c) Tuple Response Time with Window Size

Fig. 5. Average tuple response time under perfect prediction and different cases of mis-prediction.

this phase, the reduction in communication cost dominates and results in decreased response time. However, as V continues to increase, the reduction in communication cost diminishes while queue backlogs keep accumulating. Consequently, the increase in queueing delay dominates and induces the growth in response time.

Second, compared to perfect prediction, those five schemes generally incur a longer response time. Such extra response time is due to mis-prediction with both false positive and true negative. In particular, false-alarmed tuples may be pre-served by the systems and take up the queue backlogs, incurring longer queueing delays for the tuples that really need processing; meanwhile, miss-detected tuples take no benefits from pre-serving and hence their response time will not be shortened accordingly. In the meantime, compared to the case without prediction (*No Prediction*), the five forecasting schemes, although with mis-prediction, still benefit from pre-serving tuples by exploiting limited future information, by an up to 16.5% reduction in response time. However, such benefit vanishes as the value of parameter V increases and exceeds 20. The reason is that, for these forecasting schemes, larger values of V incur greater queue backlog sizes and more mis-predicted tuples, whose accumulation further prolongs the queueing delay and hence longer response time than the non-prediction case.

Insight: The above results reveal two sides of predictive scheduling. On the upside, exploiting future information does help to shorten tuple response time. The downside is that, because of the unavoidable mis-prediction, the benefit of predictive scheduling will be gradually offset by the loss in terms of longer queueing delays, which are mainly caused by accumulated mis-predicted tuples with excessively leveraged future information, *e.g.*, with a larger lookahead window. In practice, to enjoy the benefits brought by predictive scheduling, the balance must be carefully treated.

Average response time vs. lookahead window size W: In Figure 5 (c), we further consider two extreme cases of mis-prediction. One is when all actual tuple arrivals fail to be predicted, denoted by *All-True-Negative*. In fact, *All-True-Negative* is equivalent to the case without prediction since no tuples are predicted to arrive. The other is when the actual arrivals are correctly predicted, but also with some extra arrivals falsely alarmed. We denote such a case by *False-Positive (x)*, where x is the average number of false-positive tuples. Note that any case of mis-prediction is equivalent to some superposition of these two extremes. By fixing $V = 1$ and varying the value of x from 10 to 30, we acquire the following results.

First, we see that, with perfect prediction, POTUS effectively reduces tuple response time down to near-zero, as the prediction window size grows from 0 to 10. *All-True-Negative*, as anticipated, remains constant in response time. Meanwhile, *False-Positive* exhibits various performances with different values of x. In particular, when $x = 10$, the response time declines as the window size grows from 0 to 2, and goes up but remains lower than *All-True-Negative* before the window size reaching 6. From Figure 5 (b), we infer that mis-prediction, especially false-positive requests, contribute to the increase in queueing delay and thus prolongs response time. Such reduction in response time actually attributes to the advantage of predictive scheduling, *i.e.*, by exploiting surplus system resources in present time slot to pre-serve future requests and achieving more balanced workloads in temporal dimension. Nonetheless, as the window size continues to increase from 6 to 10, more and more false-positive requests will be admitted and pre-served by the system. Finally, the response time is dominated by the ever-increasing queueing delay, while the benefit of predictive scheduling vanishes. By increasing the number of false-positive tuples from 10 to 30, predictive scheduling no longer conduces to shortening response time. Instead, with excessively pre-served false-positive tuples, it results in even longer response time.

Insight: On one hand, even with mis-prediction, short-term forecasting is still helpful to shorten tuple response time. On the other hand, to exploit the power of predictive scheduling, one should adopt the forecasting scheme with a small variance in the estimate of tuple arrivals, so that false-positive arrivals will not offset the benefit of reduced response time.

V. Conclusion

In this paper, we studied the problem of tuple scheduling while conducting systematic investigation of the benefits of predictive scheduling to Apache Heron. We proposed *POTUS*, an online and efficient scheme that schedules tuples with pre-service in a distributed manner. Theoretical analysis and simulations verified the effectiveness of POTUS against the state-of-the-art schemes, with a provably near-optimal system cost in computationally efficient fashion. Further, with mild-value of future information, POTUS significantly reduces response time, even in the presence of mis-prediction.

REFERENCES

[1] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik, "Aurora: a new model and architecture for data stream management," *the VLDB Journal*, vol. 12, no. 2, pp. 120–139, 2003.

[2] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang, "Niagaracq: A scalable continuous query system for internet databases," in *Proceedings of ACM SIGMOD*, 2000.

[3] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina *et al.*, "The design of the borealis stream processing engine." in *Proceedings of Cidr*, 2005.

[4] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari, "S4: Distributed stream computing platform," in *Proceedings of IEEE ICDMW*, 2010.

[5] S. A. Noghabi, K. Paramasivam, Y. Pan, N. Ramesh, J. Bringhurst, I. Gupta, and R. H. Campbell, "Samza: stateful scalable stream processing at linkedin," in *Proceedings of the VLDB Endowment*, 2017.

[6] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham *et al.*, "Storm@twitter," in *Proceedings of ACM SIGMOD*, 2014.

[7] S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. M. Patel, K. Ramasamy, and S. Taneja, "Twitter heron: Stream processing at scale," in *Proceedings of ACM SIGMOD*, 2015.

[8] "Heron documentation," https://apache.github.io/incubator-heron/docs.

[9] A. Floratou, A. Agrawal, B. Graham, S. Rao, and K. Ramasamy, "Dhalion: self-regulating stream processing in heron," *Proceedings of the VLDB Endowment*, 2017.

[10] J. Xu, Z. Chen, J. Tang, and S. Su, "T-storm: Traffic-aware online scheduling in storm," in *Proceedings of IEEE ICDCS*, 2014.

[11] S. Chintapalli, D. Dagit, B. Evans, R. Farivar, T. Graves, M. Holderbaugh, Z. Liu, K. Nusbaum, K. Patil, B. J. Peng *et al.*, "Benchmarking streaming computation engines: Storm, flink and spark streaming," in *Proceedings of IEEE IPDPSW*, 2016.

[12] "Netflix adds download functionality," https://technology.ihs.com/586280/netflix-adds-download-support.

[13] L. Huang, S. Zhang, M. Chen, and X. Liu, "When backpressure meets predictive scheduling," *IEEE/ACM TON*, vol. 24, no. 4, pp. 2237–2250, 2016.

[14] J. D. Little, "A proof for the queuing formula," *Operations Research*, vol. 9, no. 3, pp. 383–387, 1961.

[15] M. J. Neely, "Stochastic network optimization with application to communication and queueing systems," *Synthesis Lectures on Communication Networks*, vol. 3, no. 1, pp. 1–211, 2010.

[16] "Streaming pipelines in kubernetes using apache pulsar, heron and bookkeeper," https://dwz.cn/3JmJ2nS1.

[17] A. Singla, C.-Y. Hong, L. Popa, and P. B. Godfrey, "Jellyfish: Networking data centers, randomly." in *Proceedings of USENIX NSDI*, 2012.

[18] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," in *Proceedings of ACM SIGCOMM*, 2008.

[19] B. Peng, M. Hosseini, Z. Hong, R. Farivar, and R. Campbell, "R-storm: Resource-aware scheduling in storm," in *Proceedings of AMC*, 2015.

[20] T. Benson, A. Akella, and D. A. Maltz, "Network traffic characteristics of data centers in the wild," in *Proceedings of ACM SIGCOMM*, 2010.

[21] C. K. Chui and G. Chen, *Kalman Filtering, 5th edition*. Springer, 2017.

[22] S. J. Taylor and B. Letham, "Forecasting at scale," *The American Statistician*, vol. 72, no. 1, pp. 37–45, 2018.

[23] G. E. Box, G. M. Jenkins, G. C. Reinsel, and G. M. Ljung, *Time series analysis: forecasting and control*. John Wiley & Sons, 2015.

APPENDIX-A
PROBLEM TRANSFORMATION

To solve problem (13), we adopt the Lyapunov optimization technique [15]. First, we define the quadratic Lyapunov function as

$$L(\boldsymbol{Q}(t)) \triangleq \frac{1}{2}\left\{\sum_{i\in\mathcal{I}}\left[Q_i^{(\text{in})}(t)\right]^2+\beta\sum_{i\in\mathcal{I}}\sum_{\eta\in\mathcal{N}_i^{(\text{suc})}}\left[Q_{i,\eta}^{(\text{out})}(t)\right]^2.\right\} \tag{16}$$

Then we define the Lyapunov drift for two consecutive time slots as

$$\Delta(\boldsymbol{Q}(t)) \triangleq \mathbb{E}\left\{L(\boldsymbol{Q}(t+1))-L(\boldsymbol{Q}(t))\middle|\boldsymbol{Q}(t)\right\} \tag{17}$$

, which measures the conditional expected successive change in queues' congestion state. To avoid overloading any queue backlogs in the system, it is desirable to make the difference as low as possible. However, striving for small queue backlogs may incur considerable communication cost and computation cost. Hence, we should jointly consider both queueing stability and the consequent system costs. Given routing decision $\boldsymbol{R}(t)$, we define the drift-plus-penalty function as

$$\begin{aligned}\Delta_V(\boldsymbol{Q}(t)) \triangleq\ & \mathbb{E}\left\{L(\boldsymbol{Q}(t+1))-L(\boldsymbol{Q}(t))\middle|\boldsymbol{Q}(t)\right\}\\ &+V\mathbb{E}\left\{\Theta\left(\boldsymbol{R}(t)\right)\middle|\boldsymbol{Q}(t)\right\},\end{aligned} \tag{18}$$

where V is a positive constant that determines the balance between queueing stability and minimizing total system costs.

Next, we show how problem (13) is transformed into (43) in detail. According to (18), the drift-plus-penalty term is

$$\begin{aligned}&\Delta_V(\boldsymbol{Q}(t))\\ =\ &\mathbb{E}\left\{L(\boldsymbol{Q}(t+1))-L(\boldsymbol{Q}(t))+V\cdot\Theta(\boldsymbol{R}(t))\middle|\boldsymbol{Q}(t)\right\}.\end{aligned} \tag{19}$$

Remind the definitions of quadratic Lyapunov function in (16), we expand $\Delta_V(\boldsymbol{Q}(t))$

$$\begin{aligned}&\Delta_V(\boldsymbol{Q}(t))\\ &=\frac{1}{2}\mathbb{E}\left\{\sum_{i\in\mathcal{I}^{(k)}}\left[\left(Q_i^{(\text{in})}(t+1)\right)^2-\left(Q_i^{(\text{in})}(t)\right)^2\right]\middle|\boldsymbol{Q}(t)\right\}\\ &+\frac{\alpha}{2}\mathbb{E}\left\{\sum_{i\in\mathcal{I}}\sum_{\eta\in\mathcal{N}_i^{(\text{suc})}}\left[\left(Q_{i,\eta}^{(\text{out})}(t+1)\right)^2-\left(Q_{i,\eta}^{(\text{out})}(t)\right)^2\right]\right.\\ &\left.\middle|\boldsymbol{Q}(t)\right\}\\ &+\mathbb{E}\left\{V\Theta(\boldsymbol{R}(t))\middle|\boldsymbol{Q}(t)\right\}.\end{aligned} \tag{20}$$

To derive the upper bound of (20), we show the following inequalities for any non-negative numbers a, b and c as follows

$$\begin{aligned}\left[(a+b-c)^+\right]^2 &\le (a+b-c)^2\\ &= a^2+b^2+c^2+2a(b-c)-2bc\\ &\le a^2+b^2+c^2+2a(b-c),\end{aligned} \tag{21}$$

$$\begin{aligned}\left[(a-c)^+ + b\right]^2 &\le (a-c)^2+b^2+2b(a-c)^+\\ &= a^2+c^2-2ac+b^2+2b(a-c)^+\\ &\le a^2+c^2-2ac+b^2+2ba\\ &= a^2+b^2+c^2+2a(b-c),\end{aligned} \tag{22}$$

where the last inequality in (22) holds since for $a, c \ge 0$,

$$a \ge 0 \text{ and } a \ge a-c, \tag{23}$$

and thus $a \ge \max\{a-c, 0\} = [a-c]^+$. Thereby, with the queueing update equations (6)-(8), we obtain

1) For spout instance i, regarding its output queue backlog

to component $\eta \in \mathcal{N}_i^{(\text{suc})}$, we have

$$\begin{aligned}
&\left[Q_{i,\eta}^{(\text{out})}(t+1)\right]^2 \\
&= \left\{ \left[Q_{i,\eta}^{(\text{out})}(t) - \sum_{i' \in \mathcal{I}_\eta^{(\text{N})}} R_{i,i'}(t) \right] + \lambda_{i,\eta}(t+W_i+1) \right\}^2 \\
&\leq \left[Q_{i,\eta}^{(\text{out})}(t)\right]^2 + \left[\sum_{i' \in \mathcal{I}_\eta^{(\text{N})}} R_{i,i'}(t) \right]^2 + \left[\lambda_{i,\eta}(t+W_i+1)\right]^2 \\
&\quad + 2Q_{i,\eta}^{(\text{out})}(t) \left[\lambda_{i,\eta}(t+W_i+1) - \sum_{i' \in \mathcal{I}_\eta^{(\text{N})}} R_{i,i'}(t) \right].
\end{aligned} \tag{24}$$

2) For bolt instance i, regarding its input queue, we have

$$\begin{aligned}
&\left[Q_i^{(\text{in})}(t+1)\right]^2 \\
&= \left\{ \left[Q_i^{(\text{in})}(t) + \sum_{i' \in \mathcal{I}_i^{(\text{pre})}} R_{i',i}(t) - \mu_i(t) \right]^+ \right\}^2 \\
&\leq \left[Q_i^{(\text{in})}(t)\right]^2 + \left[\sum_{i' \in \mathcal{I}_i^{(\text{pre})}} R_{i',i}(t) \right]^2 + \left[\mu_i(t)\right]^2 \\
&\quad + 2Q_i^{(\text{in})}(t) \left[\sum_{i' \in \mathcal{I}_i^{(\text{pre})}} R_{i',i}(t) - \mu_i(t) \right].
\end{aligned} \tag{25}$$

3) For bolt instance i, regarding its output queue backlog to component $\eta \in \mathcal{N}_i^{(\text{suc})}$, we have

$$\begin{aligned}
&\left[Q_{i,\eta}^{(\text{out})}(t+1)\right]^2 \\
&= \left\{ \left[Q_{i,\eta}^{(\text{out})}(t) - \sum_{i' \in \mathcal{I}_\eta^{(\text{N})}} R_{i,i'}(t) \right] + \nu_{i,\eta}(t) \right\}^2 \\
&\leq \left[Q_{i,\eta}^{(\text{out})}(t)\right]^2 + \left[\sum_{i' \in \mathcal{I}_\eta^{(\text{N})}} R_{i,i'}(t) \right]^2 + \left[\nu_{i,\eta}(t)\right]^2 \\
&\quad + 2Q_{i,\eta}^{(\text{out})}(t) \left[\nu_{i,\eta}(t) - \sum_{i' \in \mathcal{I}_\eta^{(\text{N})}} R_{i,i'}(t) \right].
\end{aligned} \tag{26}$$

We define $D_{\max}$ as the maximum in-degree and out-degree of components, $I_{\max}$ as the maximum number of instances for any component, and $\gamma_{\max}$ as the maximum transmission capacity in a time slot for one instance. Also remind the boundedness of all the tuple arrivals, service capacities of instances. In time t, for spout instance i and its succeeding component $\eta \in \mathcal{N}_i^{(\text{suc})}$,

$$\lambda_{i,\eta}(t+W_i+1) \leq \lambda_{\max}, \tag{27}$$

$$\begin{aligned}
\sum_{i' \in \mathcal{I}_\eta^{(\text{N})}} R_{i,i'}(t) &\leq Q_{i,\eta}^{(\text{out})}(t) \\
&\leq \sum_{w=0}^{W_i} \lambda_{i,\eta}(t+w) \\
&\leq (W_{\max}+1) \cdot \lambda_{\max}.
\end{aligned} \tag{28}$$

For bolt instance i, regarding its input queue,

$$\mu_i(t) \leq \mu_{\max}. \tag{29}$$

Since it has at most $D_{\max} \cdot I_{\max}$ preceding instances, and for each $i' \in \mathcal{I}_i^{(\text{pre})}$,

$$R_{i',i}(t) \leq \sum_{i'' \in \mathcal{I}_{i'}^{(\text{suc})}} R_{i',i''}(t) \leq D_{\max} I_{\max} \gamma_{\max}. \tag{30}$$

Regarding its output queue to component $\eta \in \mathcal{N}_i^{(\text{suc})}$,

$$\nu_{i,\eta}(t) \leq \nu_{\max}, \tag{31}$$

and

$$\sum_{i' \in \mathcal{I}_\eta^{(\text{N})}} R_{i,i'}(t) \leq \gamma_i \leq \gamma_{\max} \tag{32}$$

Then by injecting (24) – (26) into (20) and applying the above bounds, we obtain

$$\begin{aligned}
&\Delta_V(\boldsymbol{Q}(t)) \\
&\leq \frac{1}{2} \cdot |\mathcal{I}| \cdot \left[\left(D_{\max} I_{\max} \gamma_{\max}\right)^2 + \left(\mu_{\max}\right)^2 \right] \\
&+ \frac{\beta}{2} \cdot |\mathcal{I}| \cdot D_{\max} \cdot \left[\left(W_{\max}+1\right)^2 \cdot \left(\lambda_{\max}\right)^2 + \left(\lambda_{\max}\right)^2 \right] \\
&+ \frac{\beta}{2} \cdot |\mathcal{I}| \cdot D_{\max} \cdot \left[\left(\nu_{\max}\right)^2 + \left(\gamma_{\max}\right)^2 \right] \\
&+ \mathbb{E}\left\{ \sum_{i \in \mathcal{I}} \sum_{i' \in \mathcal{I}_i^{(\text{pre})}} Q_i^{(\text{in})}(t) \left[R_{i',i}(t) - \mu_i(t) \right] \middle| \boldsymbol{Q}(t) \right\} \\
&+ \alpha \mathbb{E}\left\{ \sum_{i \in \mathcal{I}^{(\text{spout})}} \sum_{\eta \in \mathcal{N}_i^{(\text{suc})}} Q_{i,\eta}^{(\text{out})}(t) \left[\lambda_{i,\eta}(t+W_i+1) - \sum_{i' \in \mathcal{I}_\eta^{(\text{N})}} R_{i,i'}(t) \right] \middle| \boldsymbol{Q}(t) \right\} \\
&+ \alpha \mathbb{E}\left\{ \sum_{i \in \mathcal{I}^{(\text{bolt})}} \sum_{\eta \in \mathcal{N}_i^{(\text{suc})}} Q_{i,\eta}^{(\text{out})}(t) \left[\nu_{i,\eta}(t) - \sum_{i' \in \mathcal{I}_\eta^{(\text{N})}} R_{i,i'}(t) \right] \middle| \boldsymbol{Q}(t) \right\} \\
&+ \mathbb{E}\left\{ V\Theta(\boldsymbol{R}(t)) \middle| \boldsymbol{Q}(t) \right\}
\end{aligned} \tag{33}$$

$$\begin{aligned}
&\Delta_V(\boldsymbol{Q}(t)) \\
&\leq \frac{1}{2} \cdot |\mathcal{I}| \cdot \left[\left(D_{\max} I_{\max} \gamma_{\max}\right)^2 + \left(\mu_{\max}\right)^2 \right] \\
&+ \frac{\beta}{2} \cdot |\mathcal{I}| \cdot D_{\max} \cdot \left[\left(W_{\max}+1\right)^2 \cdot \left(\lambda_{\max}\right)^2 + \left(\lambda_{\max}\right)^2 \right] \\
&+ \frac{\beta}{2} \cdot |\mathcal{I}| \cdot D_{\max} \cdot \left[\left(\nu_{\max}\right)^2 + \left(\gamma_{\max}\right)^2 \right] \\
&+ \mathbb{E}\left\{ \sum_{i \in \mathcal{I}} \sum_{i' \in \mathcal{I}_i^{(\text{pre})}} Q_i^{(\text{in})}(t) \left[R_{i',i}(t) - \mu_i(t) \right] \middle| \boldsymbol{Q}(t) \right\} \\
&+ \alpha \mathbb{E}\left\{ \sum_{i \in \mathcal{I}^{(\text{spout})}} \sum_{\eta \in \mathcal{N}_i^{(\text{suc})}} Q_{i,\eta}^{(\text{out})}(t) \left[\lambda_{i,\eta}(t+W_i+1) - \sum_{i' \in \mathcal{I}_\eta^{(\text{N})}} R_{i,i'}(t) \right] \middle| \boldsymbol{Q}(t) \right\} \\
&+ \alpha \mathbb{E}\left\{ \sum_{i \in \mathcal{I}^{(\text{bolt})}} \sum_{\eta \in \mathcal{N}_i^{(\text{suc})}} Q_{i,\eta}^{(\text{out})}(t) \left[\nu_{i,\eta}(t) - \sum_{i' \in \mathcal{I}_\eta^{(\text{N})}} R_{i,i'}(t) \right] \middle| \boldsymbol{Q}(t) \right\} \\
&+ \mathbb{E}\left\{ V\Theta(\boldsymbol{R}(t)) \middle| \boldsymbol{Q}(t) \right\}
\end{aligned} \tag{34}$$

where $\mathcal{I}^{(\text{spout})}$ and $\mathcal{I}^{(\text{bolt})}$ are the set of all spout and bolt instances, respectively. Next, by defining a constant B as

$$\begin{aligned}
B \triangleq\ & \frac{1}{2} \cdot |\mathcal{I}| \cdot \left[\left(D_{\max} I_{\max} \gamma_{\max}\right)^2 + \left(\mu_{\max}\right)^2 \right] \\
+\ & \frac{\beta}{2} \cdot |\mathcal{I}| \cdot D_{\max} \cdot \left[\left(W_{\max}+1\right)^2 \cdot \left(\lambda_{\max}\right)^2 + \left(\lambda_{\max}\right)^2 \right] \\
+\ & \frac{\beta}{2} \cdot |\mathcal{I}| \cdot D_{\max} \cdot \left[\left(\nu_{\max}\right)^2 + \left(\gamma_{\max}\right)^2 \right]
\end{aligned} \tag{35}$$

By substituting (35) into (34) and canceling the terms that are irrelevant to the decision variables $\boldsymbol{R}(t)$ (defined as $C(\boldsymbol{Q}(t))$),

we obtain

$$\begin{aligned}
&\Delta_V(\boldsymbol{Q}(t)) \\
&\le B + C(\boldsymbol{Q}(t)) + \\
&\mathbb{E}\left\{ -\alpha \sum_{i\in\mathcal{I}} \sum_{\eta\in\mathcal{N}_i^{(\text{suc})}} \sum_{i'\in\mathcal{I}_\eta^{(\text{N})}} Q_{i,\eta}^{(\text{out})}(t) R_{i,i'}(t) \middle| \boldsymbol{Q}(t) \right\} \\
&+\mathbb{E}\left\{ \sum_{i\in\mathcal{I}} \sum_{i'\in I_i^{(\text{pre})}} Q_i^{(\text{in})}(t) R_{i',i}(t) \middle| \boldsymbol{Q}(t) \right\} \\
&+V\mathbb{E}\left\{ \Theta(\boldsymbol{R}(t)) \middle| \boldsymbol{Q}(t) \right\}
\end{aligned} \tag{36}$$

Rearranging the term, we obtain

$$\begin{aligned}
\Delta_V(\boldsymbol{Q}(t)) \le\ & B + C(\boldsymbol{Q}(t)) \\
&+\mathbb{E}\left\{ \sum_{i\in\mathcal{I}} \sum_{i'\in\mathcal{I}_i^{(\text{suc})}} l_{i,i'}(t) R_{i,i'}(t) \middle| \boldsymbol{Q}(t) \right\}
\end{aligned} \tag{37}$$

where

$$l_{i,i'}(t) \triangleq V \sum_{\kappa,\kappa'\in\mathcal{C}} X_{i,\kappa} X_{i',\kappa'} U_{\kappa,\kappa'}(t) + Q_{i'}^{(\text{in})}(t) - \beta Q_i^{(\text{out})}(t). \tag{38}$$

By minimizing the upper bound of the drift-plus-penalty expression in (37), we can minimize the long-term time average of total system cost while stabilizing all processing queues. To transform the minimization of the above bound to minimization of the objective in (13), we have the following statement. We denote the objective function at time slot t by $J_t(\boldsymbol{R})$ with decision $\boldsymbol{R}$, and its respective optimal solutions by $\boldsymbol{R}^*$.

$$J_t(\boldsymbol{R}) \triangleq \sum_{i\in\mathcal{I}} \sum_{i'\in\mathcal{I}_i^{(\text{suc})}} l_{i,i'}(t) R_{i,i'} \tag{39}$$

and

$$\boldsymbol{R}^* \in \arg\min_{\boldsymbol{R}} J_t(\boldsymbol{R}) \tag{40}$$

Hence, for any other feasible scheduling decisions $\boldsymbol{R}$ made during time slot t, we have

$$J_t(\boldsymbol{R}) \ge J_t(\boldsymbol{R}^*) \tag{41}$$

By taking the conditional expectation on both sides conditional on $\boldsymbol{Q}(t)$, we have

$$\mathbb{E}\left\{ J_t(\boldsymbol{R}) \middle| \boldsymbol{Q}(t) \right\} \ge \mathbb{E}\left\{ J_t(\boldsymbol{R}^*) \middle| \boldsymbol{Q}(t) \right\} \tag{42}$$

for any feasible $\boldsymbol{R}$.

Inequality (42) reveals that $\boldsymbol{R}^*$ minimizes the conditional expectation of $J_t(\boldsymbol{R})$, thusly minimizing the upper bound of drift-plus-penalty in (37). In such a way, instead of directly solving the long-term stochastic optimization problem (13), we can opportunistically choose a feasible association to solve the following problem during each time slot.

$$\begin{aligned}
\underset{\boldsymbol{R}}{\text{Minimize}} \quad & J_t(\boldsymbol{R}) \\
\text{Subject to} \quad & 0 \le \sum_{c'\in\eta_{c(i)}} \sum_{i'\in\mathcal{I}_{c'}^{(k)}} R_{i,i'} \le \tau_i \ \forall\ i \in \mathcal{I}^{(k)}, \\
& \sum_{i'\in\mathcal{I}_{c'}^{(k)}} R_{i,i'} \le H_{i,c'}^{(\text{out})}(t) \ \forall\ i \text{ and } c' \in \eta_{c(i)}.
\end{aligned} \tag{43}$$

APPENDIX-B
PROOF

We assume there is an S-only algorithm achieves optimal time-average total cost (infimum) Θ^* with action $\tilde{\boldsymbol{R}}(t)$ for $t = \{0, 1, 2, \cdots\}$.

From [15], we know that to ensure queue stability, the expectation of arrival rate must be no more than the expectation of service rate, with a difference of $\epsilon \ge 0$.

Denote $\boldsymbol{R}'(t)$ as the decisions over time, and $\Theta'(t)$ as the corresponding communication cost given by POTUS. According to (36), the one-slot drift-plus-penalty here is

$$\begin{aligned}
&\mathbb{E}\left\{ L(\mathbf{H}(t+1)) - L(\mathbf{H}(t)) \middle| \mathbf{H}(t) \right\} + V\mathbb{E}\left\{ \Theta'(t) \middle| \mathbf{H}(t) \right\} \\
&\le B + V\mathbb{E}\left\{ \Theta^*(t) \middle| \mathbf{H}(t) \right\} - \epsilon\mathbb{E}\left\{ h(t) \middle| \mathbf{H}(t) \right\}
\end{aligned} \tag{44}$$

Taking expectation over both sides, we obtain

$$\begin{aligned}
&\mathbb{E}\left\{ L(\mathbf{H}(t+1)) \right\} - \mathbb{E}\left\{ L(\mathbf{H}(t)) \right\} + V\mathbb{E}\left\{ \Theta'(t) \right\} \\
&\le B + V\mathbb{E}\left\{ \Theta^*(t) \right\} - \epsilon\mathbb{E}\left\{ h(t) \right\}
\end{aligned} \tag{45}$$

Summing over $t = \{0, 1, \cdots, T-1\}$, we have

$$\begin{aligned}
&\mathbb{E}\left\{ L(\mathbf{H}(T-1)) \right\} - \mathbb{E}\left\{ L(\mathbf{H}(0)) \right\} + V\mathbb{E}\left\{ \sum_{t=0}^{T-1} \Theta'(t) \right\} \\
&\le BT + V\mathbb{E}\left\{ \sum_{t=0}^{T-1} \Theta^*(t) \right\} - \epsilon\mathbb{E}\left\{ h(t) \right\}
\end{aligned} \tag{46}$$

Using the derived inequality (46), we show the two performance bounds in *Theorem 1*.

1) First, dividing both sides of (46) by VT, rearranging items and neglecting non-positive quantities in right side, we obtain

$$\begin{aligned}
&\frac{1}{T}\sum_{t=0}^{T-1} \mathbb{E}\left\{ \Theta'(t) \right\} \\
&\le \frac{B}{V} + \frac{\mathbb{E}\{L(\boldsymbol{Q}(0))\}}{VT} + \frac{1}{T}\sum_{t=0}^{T-1} \mathbb{E}\left\{ \Theta^*(t) \right\}
\end{aligned} \tag{47}$$

As $T \to \infty$, we have

$$\begin{aligned}
&\limsup_{T\to\infty} \frac{1}{T}\sum_{t=0}^{T-1} \mathbb{E}\left\{ (\Theta'(t)) \right\} \\
&\le \limsup_{T\to\infty} \frac{1}{T}\sum_{t=0}^{T-1} \mathbb{E}\left\{ (\Theta^*(t)) \right\} + \frac{B}{V}
\end{aligned} \tag{48}$$

After simplification, we obtain

$$\bar{\Theta}' \le \bar{\Theta}^* + \frac{B}{V}. \tag{49}$$

2) Similarly, by dividing both sides of (46) by ϵT, we get

$$\begin{aligned}&\frac{1}{T}\sum_{t=0}^{T-1}\mathbb{E}\left\{h(t)\right\}\\&\leq\frac{B}{\epsilon}+\frac{\mathbb{E}\{L(\mathbf{H}(0))\}}{\epsilon T}+\frac{V\sum_{t=0}^{T-1}\mathbb{E}\left\{\Theta^*(t)\right\}}{\epsilon T}.\end{aligned}\tag{50}$$

As $T\to\infty$, we obtain

$$\bar{h}\leq\frac{V\bar{\Theta}^*}{\epsilon}+\frac{B}{\epsilon}.\tag{51}$$

■