On Optimal Control of Computational Streams

Andrey Khorlin and K. Mani Chandy
Computer Science 256-80
California Institute of Technology
Pasadena, California 91125 USA
{akhorlin, mani}@cs.caltech.edu

Abstract

Stream processing systems receive continuous streams of messages with relatively raw information and produce streams of messages with processed information. The utility of a stream-processing system depends, in part, on the accuracy and timeliness of the output. Streams in complex event processing systems are processed on distributed systems; several steps are taken on different processors to process each incoming message, and messages may be enqueued between steps. This paper explores the problem of distributed dynamic control of streams to optimize the total utility provided by the system. A system can be controlled using central control or distributed control. In the former case a single central controller maintains the state of the entire system and controls operation of all processors. In distributed control systems, each processor controls itself based on its state and information from other processors. A challenge of distributed control is that timeliness of output depends only on the total end-to-end time and is otherwise independent of the delays at each separate processor whereas the controller for each processor takes action to control only the steps on that processor and cannot directly control the entire network.

1 Introduction

Stream-processing systems process, transform, correlate, and react to rapidly arriving streams of messages. This functionality allows an on-demand enterprise to quickly react to rapidly developing opportunities and threats. These systems require a great degree of adaptability because many aspects of the system change including the relative importance of different streams, the rates at which messages arrive, processing requirements at each step, queue sizes and numbers of event sources. In many cases, the amount of processing and data requires that the system be distributed over a grid of servers [8].

The hardware resources of the system are represented by an undirected graph where the vertices represent processors and the edges represent communication links. The steps of stream processing are represented by a directed graph where each vertex represents a step and the edges represent flows of information between steps. Given a mapping of the graph representing the logical flow-structure of a stream processing system to the graph representing the physical resource structure of the underlying distributed system [38], the problem is to design distributed control policies to optimize the overall utility of the system.

Control theory offers tools that help meet these requirements. Issues of robustness and adaptability have been dealt with for a long time in engineering and control theory. For example, Mars rovers were designed to have robust performance in a foreign environment of another planet [20] and DARPA grand challenge vehicles were built to find their way in the Mojave desert [17]. The key tools for the design of such systems have been developed in the area of feedback control of dynamic systems. Recently distributed control theory has helped explain the behavior of different TCP protocols [29].

In this paper, we outline algorithms for scheduling streams in a distributed stream processing system. In such a system, a single data-stream processing application is distributed among several servers creating an overlay network. At each server, a decision must be made on what jobs must be processed when. The selection decision depends on the cost functions defined for each flow. These functions correspond to business application constraints. A change in scheduling policy at a server in a distributed system can cause changes in backlogs at that server and in servers downstream. The impact of changes in policies can be complex when the system has a number of streams.

Specifying a policy for a single central controller that has all the information about the entire system is easier than specifying policies that coordinate multiple controllers for each server or group of servers. A single central controller is not feasible if the number of servers is large because get-
The rest of the paper is organized as follows. Section two provides problem definition. Section three outlines a scheduling algorithm based on process sharing scheme in a single and multi-server environments. Section four describes a marginal cost-based scheduling algorithm. Section five discusses experimental results. Finally, we provide conclusions, discussion of the future research and related work in sections six, seven and eight, respectively.

## 2 Problem Definition

In this paper, we assume that the data-stream processing application consists of several user-defined, computation flows. Each computation flow is viewed as a direct acyclic graph with each node representing an indivisible unit of computation. Moreover, due to computational intensity of these flows, the flows are distributed between a set of servers who are connected in a topology. Our system assumptions are similar to the one presented in [21], [1] and [5]. For the purposes of this paper, the allocation of flows to servers is assumed to be given.

After the computation is mapped onto a topology of servers, a single server may contain several processing steps of different flows. These flows compete for local resources. We assume that message arrival is a random process. This leads to messages being queued up at each server. Control policies determine which messages to process next given estimates about the state of the total system. The mathematical models we build assume that the arrival process is Poisson and that service times are exponentially distributed.

We adopt a notion of a quality of service functions (QoS), \( q_i(x) \), (similar to [1]). Each computation flow has a QoS function assigned to it. This function maps the delay of the events produced by this flow to a measure of cost. The delay for resulting events is measured as follows. A message entering the system is timestamped. Each operator in a data flow takes several input messages and produces zero or more messages as defined in [4], [21] or [1]. An output message carries the highest timestamp from all messages used to create it. When final output message arrives at a sink, the difference between current time and the timestamp is measured. The different types of QoS functions used in our work are shown in table 1.

<table>
<thead>
<tr>
<th>QoS Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( q(x) = a \cdot x + b )</td>
<td>Linear QoS</td>
</tr>
<tr>
<td>( q(x) = \log_a (b \cdot x + c) )</td>
<td>Concave QoS</td>
</tr>
<tr>
<td>( q(x) = -\frac{x}{b} )</td>
<td>Convex QoS</td>
</tr>
<tr>
<td>( q(x) = \frac{w}{\log(1+e^{b\cdot x})+c} )</td>
<td>Sigmoidal QoS</td>
</tr>
</tbody>
</table>

Table 1. Types of QoS functions

Each of these types represents a different business scenario. For example, the linear function represents a scenario when the cost of delay is proportional to delay. Therefore, minimizing the cost would result in proportional minimization of the delay for all messages in the stream. The concave function represents the case where the marginal cost of delay diminishes, the longer a message is delayed. The convex function represents the case where the marginal cost of delay increases the longer a message is delayed; increasing delay of messages that have been in the system for some time results in stiff penalties. The sigmoid function is a continuous equivalent of a step function, which represents the case when message with certain delay up to time \( x \) are not penalized; however, if message passes certain deadline, then the penalty becomes substantial. In general, we assume that the QoS function could be any differentiable, non-decreasing function.

In a given setup, the goal of the scheduling algorithm is to minimize a running average of cost for all data flows in the system. More formally, the goal of our scheduling algorithm is to minimize

\[
\frac{1}{|F|} \lim_{t \to \infty} \frac{\sum_{i=0}^{F} \sum_{t=0}^{N_i} q_i(d_i^t)}{N_i^t}
\]

where \( N_i^t \) is number of events received by stream \( i \) before time \( t \) and \( F \) is the set of flows where the flows are indexed \( i \). In other words, the cost function is applied to each event produced by a flow. The resulting costs are averaged out over time. This cost is called a flow cost. Then, the average of all flow costs determines the cost for the whole system.

In this paper we assume that a significant part of the total end-to-end delay is due to queuing. Because of queuing delays, a control policy that gives an increasingly large fraction of a server to processing a stream with a high cost function may increase, rather than decrease, total cost. This is because if queues have built up for this stream in downstream servers, then giving this stream more upstream resources will merely make the downstream queues larger. We can call this a “hurry up and wait” policy. On the other hand, if a stream gets too little upstream resources then downstream servers may have no messages from this stream thus increasing the end-to-end response time. The resources given to each stream at each server must be balanced across the entire network

Now, we present two approaches for QoS-based scheduling in a stream processing environment. The first approach is based on a process sharing scheme and uses centralized optimization algorithm to reduce system delay (Section 3).

---

**Note:** The above text and table were generated based on the provided image and raw text content. Any additional or missing details beyond what is visibly present in the image have been included to ensure a complete and accurate representation of the document content.
The second approach uses feedback control to provide local scheduling with information needed to properly adjust local flow priorities (Section 4).

3 Process Sharing-based Scheduling

The first algorithm, we present here is based on the processor sharing scheme. Each stream $i$ is assigned a share, $p_i \in [0,1]$ of the processor, such that $\sum_{j=0}^n p_j = 1$. Then, the percentage of the machine given to a stream $i$ is

$$\frac{p_i}{\sum_{j=0}^n p_j}$$

$m$ is a number of streams whose queue is not empty. The processor is divided only between the streams for which there are messages in that server. The processor is not kept idle in reserve for messages that may come later from a stream.

Given this processor sharing algorithm, what are the values of $p_1, \ldots, p_n$ such that expected cost for each flow is minimized? To determine these values, we represent the algorithm as a continuous-time Markov Process. The states of the process are determined by each flow’s queue sizes. The transition rates are based on the expected inter-arrival rates, expected service rates and the shares (See definition 3.1).

**Defn 3.1.** Given processor sharing algorithm on $n$ queues and given two states $S_i = [s_{i1}, \ldots, s_{in}]$ and $S_j = [s_{j1}, \ldots, s_{jn}]$ where $s_{ij}$ is a size of queue $j$ in a state $i$, the transition function $t : \{S_i, S_j\} \rightarrow R$ specifies transition rate from state $S_i$ to state $S_j$ and is defined as

$$t(S_i, S_j) = \begin{cases} 
\lambda_j & \text{if } (S_j - S_i) = \bar{1}; \\
\omega_i \mu_j & \text{if } (S_j - S_i) = -\bar{1}; \\
0 & \text{otherwise;}
\end{cases}$$

where $\omega_i = \frac{p_i}{\sum_{k=1}^n p_k}$.

Note: $(S_j - S_i)$ represents vector subtraction and $\bar{1}$ is a vector that has 1 in exactly one position and zeros everywhere else.

Once the process is defined, we can find invariant probabilities ($\pi Q = 0$). From invariant probabilities, expected queue lengths are calculated. Then, by Little’s Law, we can compute the expected delay. Since the delays are distributed nearly exponentially, we can evaluate expected cost under given shares, $p_1, \ldots, p_n$ using formula below.

$$\overline{Cost}_i = \sum_{i=0}^{\overline{D}} re^{-ni} q_i(i) \text{ where } r = \frac{1}{\overline{D}}$$

$\overline{D}$ is the average delay from Markov Model and $q_i$ is QoS function for stream $i$.

It should be noted that the evaluation of invariant probabilities is very computationally intensive since the average model under consideration consists of billions of states. For example, the number of states of the model for five queues whose size does not exceed ten is $10^5$. And the size of $Q$-matrix is $10^{52}$.

We have created a very fast, memory non-intensive approximation algorithm for invariant probability evaluation. First, we note that we could convert $Q$-matrix into $P$-matrix. $P$-matrix contains transition probabilities instead of rates.

$$p(S_i, S_j) = \frac{t(S_i, S_j)}{\sum_{k=0}^n t(S_i, S_k)}$$

Then, we can represent $P$-matrix implicitly using the formula above. The invariant probabilities can be found by performing iteration algorithm:

$$\pi_{t} = \pi_{t+1}$$

Moreover, $\pi_t$ can be represented as a segment tree because most elements of $\pi_t$ are zero and non-zero element occur in consecutive chunks. We can further reduce the size of $\pi_t$ by round elements of $\pi_t$ that are within $\epsilon$ of 0. The detailed evaluation of this algorithm together with related proofs is outside the scope of this paper and may be found in [22].

With the model that can predict the impact of parameter values on our scheduling, we can create an optimization strategy to find optimal parameters for each server in a topology. Since, the QoS function may not be just concave or convex, we use the most generic optimization scheme, genetic algorithms [41]. Our genome is a two dimensional array consisting of $p_1, \ldots, p_n$ for each server. And the genome evaluation function is our model. At run-time, the system monitors the parameters at each server (i.e. arrival and service rates). When they change by more than $\epsilon$, the re-optimization is triggered. This approach to scheduling is centralized. Currently, we are evaluating a distributed solution that performs rounds of adjustments to shares on each server until the optimum is found.

4 Marginal Cost-based Scheduling

The processor sharing algorithm performs analysis is based on expected stream statistics. However, if stream statistics change rapidly, such algorithm may not react fast enough to adjust the shares. Thus, we present an alternative scheduling scheme that makes scheduling decision at each $\Delta t$. For this algorithm, we assume that the cost of switching between tasks is negligible. In the future, our analysis could be extended to take into account this cost.

In a single-server case, our scheduling algorithm works as follows. At each $\Delta t$, the algorithm inspects all unfinished messages for all local flows. For each message, the
following metric is computed
\[ \mu_i \Delta t \frac{dq_i(x)}{dx} \bigg|_{x=T_i} \]
\(\mu_i\) is the expected service time of the message. \(T_i\) is the amount of time this message spent in the system. For messages delivered to the end users, \(T_i\), is the total delay used to assign cost to messages.

Then, for the next \(\Delta t\), the algorithm schedules the message with the highest metric value. In a single server environment, it is proven that our algorithm minimizes the expected cost. In essence, the metric computes marginal cost of delaying a message for another \(\Delta t\) amount of time. Thus, scheduling the message with highest cost, the expected cost is minimized. Detailed proofs could be found in [22].

Our algorithm may result in messages being processed out of order. In fact, in a single-server environment, we have shown that if \(\frac{dq_i(x)}{dx} \geq 0\), the scheduling policy degenerates into first come, first serve scheduling (FCFS). And if \(\frac{dq_i(x)}{dx} < 0\), the scheduling policy is last come, first serve (LCFS) [22]. In some stream processing systems, message reordering is not allowed [1]. If message reordering is not allowed, we can modify our scheduling algorithm to avoid message reordering.

The modified algorithm only selects messages from the heads of the queues and uses a different selection metric. The new metric is
\[ \mu_i \Delta t (\gamma + \phi) \]
\(\gamma\) represents the marginal cost of delay similar to the previous case.
\[ \gamma = m_c(T_i) \]
\[ m_c(t') = \frac{dq_i(x)}{dx} \bigg|_{x=t'} \Delta t \]
\(\phi\) represents the marginal cost due to the queue size. When, message in the head of the queue is delayed all messages are delayed.
\[ \phi = \delta(1, i) - \delta(2, i) \]
\[ \delta(s, i) = \sum_{j=s}^{n_i} m_c(T_j) + j\mu_i^{-1} \]
The delay for each message in the queue is proportional to the number of jobs between the current message and the head of the queue scaled by the expected service time, \(\mu_i^{-1}\). Thus, the modified algorithm not only takes into account the cost of delaying a message at the head of every queue, but also considers the size of each queue (For proofs, see [22]).

In a multi-server environment, the scheduling algorithm should be modified. Local scheduling must know future delays that the results of the local computation will experience before they reach the user. To deal with these future delays, we provide a notion of feedback information, \(\omega\) that is sent from downstream servers. This information flows in the opposite direction to data messages and contains an estimate of current delay on a server. When a server receives \(\omega\), it can adjust marginal cost computation as follows.
\[ m_c(t') = \frac{dq_i(x)}{dx} \bigg|_{x=(t'+\omega)} \Delta t \]

This distributed scheduling approach can be viewed as a feedback control system. Currently, the feedback, we are using, consists of expected service times for each flow on a server. This feedback does not take into account queuing delays accurately. Queuing delays depend on the rate using, consists of expected service times for each flow on a server. This feedback does not take into account queuing delays accurately. Queuing delays depend on the rate with which scheduling algorithm selects messages. Which in turn, changes the behavior of the algorithm. Thus, there is a feedback loop not only between a pair of servers in a network, but also between local queues and scheduling engine. In addition, the queuing delays have a tendency to change rapidly. Thus, the only meaningful way of providing feedback about them is by accumulating statistics in a sliding window. However, the size of this window impacts the scheduling and vice-versa, creating another feedback loop. All these issue complicate the development and analysis of more accurate feedback algorithm.

<table>
<thead>
<tr>
<th></th>
<th>Experiment 1</th>
<th>Experiment 2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Steam 1</td>
<td>Steam 2</td>
</tr>
<tr>
<td>(\lambda^{-1})</td>
<td>250</td>
<td>250</td>
</tr>
<tr>
<td>(\mu^{-1})</td>
<td>100</td>
<td>70</td>
</tr>
<tr>
<td>([p_1, p_2])</td>
<td>10, 4</td>
<td>4, 7</td>
</tr>
<tr>
<td>(q_{max})</td>
<td>44, 11</td>
<td>33, 13</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Experiment 1</th>
<th>Experiment 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Steam 1</td>
<td>Steam 2</td>
</tr>
<tr>
<td>(\lambda^{-1})</td>
<td>250</td>
</tr>
<tr>
<td>(\mu^{-1})</td>
<td>100</td>
</tr>
<tr>
<td>([p_1, p_2])</td>
<td>10, 4</td>
</tr>
<tr>
<td>(q_{max})</td>
<td>44, 11</td>
</tr>
</tbody>
</table>

**Table 2. Experiment Parameters**

**Figure 1. Markov chain model accuracy**

However, indirectly expected, service times feedback takes into account some of the queuing delay. Since ser-
vice times are distributed exponentially, most true service times are below the expected service time. Expected service time overestimates true service time, and thus take into account some amount of queuing delay. We have verified this hypothesis by making $\omega$ and true future service for each messages. Such change in feedback did not result in better algorithm performance [22]. We are continuing to work on adding more precise information on queuing delays and formally showing the efficacy of this feedback.

5 Experimental Results

Now we show how our algorithms perform is a single and multi-server environments. Ptolemy discrete-event simulator is used for performance evaluation [9]. We have extended Ptolemy run-time with a set of custom components that perform scheduling using the algorithms we defined.

In the first round of experiments, we show that the Markov process outline in section 3 can predict accurately the delays of our process sharing algorithm. Two flows are located on one server parameterized as stated in Table 2. The approximation is performed assuming certain limit on queue size for these two flows. With each experiment iteration, the limit is increased. One set of experiments is performed on equal pair of streams. The other had fast and slow arriving streams. The error between approximation is plotted in relation to the maximum queue size used. Both sets of experiments show that approximation becomes accurate quite fast and after queue size of 12, (half of the maximum queue size) the error falls within 10% (Figure 1).

In the second set of experiments, we run our scheduling algorithms on a single server. The server has two streams with convex QoS functions. Stream 1’s QoS is $2x^2$. And Stream 2’s is $x^3/200$. Convex QoS functions represent the case when greater delays result in greater cost. In this set of experiments, the arrival rates and service rates for both streams are kept the same and are slowly increased to raise utilization from 20% to 90%.

Figure 2 represents the advantage of our algorithms over FIFO as a function of utilization. As utilization increases, the advantage of all three algorithms grows because the queue sizes increase. However, the growth is tampering off because, as delays decrease, the marginal costs also decrease because of the convex QoS functions. As we can see, the processor sharing algorithm slightly outperforms marginal cost algorithms due to the fact that the selection of time interval, $\Delta t$, effects the accuracy of our scheduling. $\Delta t$ cannot be made too small since it will result in inefficient execution. Moreover, the reordering doesn’t impact algorithm performance because as is stated earlier, under increasing QoS function both algorithms reduce to the same scheduling policy.

In the third round of experiments, we run our scheduling in a two-server environment. The mapping of flows to servers is shown on Figure 3. Flow 2 and 3 have linear QoS, $c \times x + 100$ where $c \in [0, 25]$. Flow 1 has a sigmoidal QoS, \[1 + e^{-x/10} / 100000]. In each iteration of this experiment set, we increase the slope of linear QoS, $c$, in order to change the amount of possible trade-off between stream 1 and stream 2 and 3. In this round, we show that our algorithm perform better if feedback is present.

The results are shown on Figure 4. We note that feedback really improved the behavior of marginal cost-based scheduling due to the fact that without feedback the scheduling would delay messages on the first server long enough, that on the second server, the maximum cost is reached. With feedback, scheduling on server 1 is aware that more processing for flow 1 is still to come, it prioritizes flow 1 over flow 2. The processor sharing algorithm doesn’t perform as well for two reasons. The way we compute cost from predicted average delay introduces an error and genetic algorithm stage produces only near optimal result. Lastly, a scheduling with no reordering is also is not gaining much benefit from feedback, because with no reordering queue dynamics play larger role. Since, queuing delays are currently not captured in the feedback, the effi-
cacy is less as compared to the scheduling with reordering. Lack of reordering complicates the analysis even beyond the issues outline in section 4. More extensive performance evaluation could be found in [22].

6 Conclusion

The new generation of data-stream processing systems create new demands for system robustness, performance and adaptability. The theory of control of dynamic systems has been shown to be effective not only in the areas outside of computer science, but also for design of new networking algorithms [29]. We believe that the next generation of stream processing systems will incorporate these ideas in their design.

We have presented two scheduling algorithms. One is based on queuing theory. And the other is based on feedback for scheduling distributed data flows. The first cut of these algorithms lacks sophistication. Despite, non-complex feedback and presence of a series of simplifying assumptions behind our scheduling algorithms, they perform very well against naïve scheduling approaches such as FIFO. And we believe offer a good foundation for future development of more sophisticated approaches.

7 Future Work

In the future, we will work on developing more sophisticated feedback for our marginal-cost based scheduling algorithm with queuing delays being taken into more accurately. Moreover, we will use the formal control theory tools to prove stability and optimality of this new feedback, similar to [29]. We also will look at finding better approximations for our processor sharing scheduling algorithm, so the invariant probabilities could be computed for larger Markov models. The approximation of such large Markov models have research value in its own right.

More importantly, we will have to work on combining scheduling approaches presented here with resource allocation algorithm developed in [38]. Since, scheduling impacts performance of resource allocation, and in turn allocation determines how much advantage can be obtained at run-time, the two have to provide information feedback to each other in order to achieve optimal performance.

8 Related Work

In general, the area of scheduling is very extensive, especially, in the domains of operating systems and networking [24], [40] and [26]. Since our work is uniquely focused on distinct aspects of distributed data-stream processing systems, we discuss only the work that is directly related to scheduling in a stream processing applications. A good description of the requirements for such systems is provided in [8].

There are several stream processing systems that are being developed in academia and industry. One of such systems is SMILE (Smart-Middleware Light-Ends) that is being developed at IBM Research [21]. SMILE introduces a novel correctness guarantee that results in simplified operator semantics and fault-tolerance algorithms. Also, as part of SMILE system, a novel resource allocation algorithm is being developed, which uses queuing theory analysis. This analysis was another source of inspiration for this work. Since SMILE is a distributed stream processing system, the scheduling algorithms proposed in our work could be easily implemented in SMILE system.

Borealis is one of the first stream processing system that used QoS functions to drive system performance. Borealis is the second generation stream processing system built as an extension to Aurora and Medusa systems. Borealis allows users to define streaming computation and their QoS requirements in a graphical fashion. There are several novel algorithms proposed as part of Borealis that include scheduling and resource allocation. In [1], the design of the future Borealis system is presented. It outlines the distributed scheduling algorithm that is being developed for the next version of the system. The scheduling strategy in [1] is based on Aurora scheduling outlined in [10]. In [10], several schemes are proposed to schedule a set of query execution graphs with associated QoS constraints. Several metrics are presented in order to optimize for different parameters including the cost of execution per operator, latency and memory requirement. In case when QoS are defined, the operators are evaluated based on the expected impact on utility. Thus, the scheduling is operator-based and not flow-based. In order to reduce the overhead, scheduling decisions are made for a set of tuples rather than on per-tuple
basis.

In spirit, our approach is similar to the one described in [1] and [10], since our algorithm tries to use auxiliary information that travels in messages to predict message impact of scheduling policy on global QoS objectives. However, our framework proposes to use feedback control to make local scheduling more adaptable and to alleviate the need for several tiers of optimization algorithms proposed in [1]. Moreover, we also take into account the effect of queuing. As was shown, queuing could have a great impact on system performance and must be taken into account by the scheduling algorithm to achieve good performance. In the future, it would interesting to compare different scheduling strategies and incorporate other types of QoS functions presented in [1] into our framework and use control theory to design a more robust system.

Event correlation engines are a special type of stream processing systems that are very popular in enterprise computing [14], [2]. Active Middleware technology (Amit) is one of leading correlation engines [2]. Amit defines a rich and extensive language and associated run-time for event detection and correlation. In the future, for a correlation system like Amit, the distributed implementation of run-time could face the same issues of control of queuing delays as outlined in this work.

Infopipes at Georgia Tech is another project, whose goal is to define and implement a data-stream processing system [23]. Infopipes provides a new abstractions that simplify development of distributed data-stream applications. A related project from Georgia Tech presents a distributed, utility-driven, resource allocation algorithm [25]. This algorithm takes into account business utility of operators in order to aggregate them and deploy them on a distributed network of servers. We envision that smart utility-driven scheduling is still performed by the run-time after allocation, since many times the re-allocation of operators may be costly.

Another stream processing system, called STREAM, is being developed by a team in Stanford [5]. STREAM proposes SQL-like language, called CQL, for querying streams. STREAM emphasizes approximations of the correct result to speed up query processing and minimize memory use. Scheduling algorithm proposed as part of STREAM focuses on memory management [3]. It does take into account latency bounds on query execution, but it is not QoS-driven as is Borealis or our algorithm. In addition, this algorithm is not designed for distributed environments, where local choices may negatively effect latency on the downstream server machines due to queuing delays.

There are many other stream processing or continuous query systems being developed [37], [16], [15], [30], [36], [13], [34], [12], [11] and [18]. Several of them such as [16] [36] and are designed specifically for network traffic monitoring domain. Some propose a general service infrastructure for development of stream processing systems [34], [18]. Many of these systems are either not distributed or use shared memory architectures. However, some of them are distributed, including [34] and [18], can benefit from our scheduling algorithms.

Lastly, Δ-data flows represent a different and novel snapshot-based paradigm for stream processing [27]. The scheduling algorithms presented here do not extend to Δ-data flows environment. In [42], several alternative algorithms for scheduling for Δ-data flows are presented.

Many of the existing stream processing and event dissemination ideas and concepts are made into real-world applications such as IBM WebSphere ESB [19], Oracle Application Server [28], BEA ESB [6], SAP [32], and TIBCO ESB [39]. Our scheduling algorithms could also be useful in the context of these systems.

There are several areas outside of stream processing that are relevant to our research. These areas include steady-state approximation of Markov chains using aggregation-disaggregation [35] and [7], and approximate mean-value analysis [31] for estimating expected queue sizes. However, the structure of our Q-matrix is not suitable for these approaches. Instead our approximation exploits the unique structure of Q-matrix to achieve limited scalability.

Work on FAST TCP, a new TCP algorithm that uses control theory to formally show stability and optimality of the congestion control algorithm could serve as an example of how we can apply control theory to scheduling in a stream processing system [29]. Generally, stochastic control and its applications in system analysis are the tools that we will continue to use in our future work on scheduling algorithms [33], [26].

References


