Control-based Scheduling in a Distributed Stream Processing System

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 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 deals with the problems of distributed dynamic control of streams to optimize the total utility provided by the system. 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.

This paper identifies key problems in distributed control and analyzes two scheduling algorithms that help in an initial analysis of a difficult problem.

1. Introduction

Stream-processing systems process, transform, correlate, and react to streams of messages. This functionality allows an on-demand enterprise to react to opportunities and threats in a timely manner. These systems adapt to changes in the relative importance of the different streams, the rates at which messages arrive, the processing requirements, the queue sizes and the numbers of the 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 streaming computation is represented by a directed graph, where each vertex is an indivisible computation step and each edge represents a flow of information between the steps. A system typically has many streams and thus the steps of stream processing are represented by a collection of graphs. The resource management problem is two-fold: (1) Map the stream processing graphs on to the resource graph, and (2) given this mapping, design distributed control policies to optimize the overall utility of the system with robust performance in an uncertain and changing environment. Stream graphs are re-mapped to the resource graph much less frequently than changes in dynamic control parameters.

Control theory offers tools that help meet these requirements. Issues of robustness and adaptability have been dealt with for decades in engineering and control design.

In this paper, we outline algorithms for scheduling streams in a distributed stream processing system. In such a system, many data-stream processing applications are distributed among several servers. 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 the 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 mapped to networks of resources, especially because of queuing effects. Using a greater fraction of a server to process a stream with a critical response time may hurt rather than help the overall utility of the system because the messages from the stream may merely get enqueued downstream. This may be thought of as a "hurry up and wait" policy. By contrast, using more of the server to lower-utility streams can help overall utility provided the downstream resources for this stream are not congested.

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 a group of servers. A central controller does

Proceedings of the IEEE Services Computing Workshops (SCW'06)
0-7695-2681-0/06 $20.00 © 2006 IEEE

not scale well with the number of servers.

The problem of distributed control of networks to maximize utility is difficult. This paper makes three preliminary contributions to this complex subject that requires much more study: (1) the paper identifies critical problems in distributed control , (2) a Markov model is proposed and analyzed for a simple case, and the difficulties of getting closed-form product-form solutions are explored, and (3) a simple distributed control algorithm based on economic concepts, such as marginal costs and benefits is presented.

We made several assumptions in the development of these algorithms so that we could analyze them and understand relationships between critical parameters. The side-affects of join and split operator semantics are not considered. The overhead of feedback is not considered. We assume that each feedback message is small. Simple models help in providing insights for the future development of advanced distributed control.

The rest of the paper is organized as follows. Section 2 presents a problem definition and identifies critical issues in distributed control of networks that show why simplifying assumptions are necessary to get tractable models. Section 3 outlines a scheduling algorithm based on a processor sharing scheme in a single and multi-server environments. Section 4 describes the marginal cost-based scheduling algorithm. Section 5 discusses experimental results. Finally, we provide conclusions, discussion of the future research and the related work in sections 6, 7 and 8, respectively.

2. Problem Definition

In this paper, we assume that the data-stream processing application consists of a set of user-defined, computation flows F. Each computation flow f_i is represented by a directed acyclic graph with each node representing an indivisible step of the computation. The flows are distributed between a set of servers with communicating channels between servers. The servers and channels are represented by a directed graph. Our system assumptions are similar to the one presented in Jin *et al.* [19], Abadi *et al.* [1] and Babu *et al.* [5]. For the purposes of this paper, the mapping of flows to servers is assumed to be given.

A single server may contain several processing steps of different flows (local flows). These local flows compete for resources. We assume that the message arrival is a non-uniform, 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.

Associated with each stream i is a *quality of service* functions (QoS), $q_i(x)$ where x is the end to end delay for messages in this stream and q_i determines the utility of a message with delay x. Generally, the greater the delay x the lower the utility; however, the manner in which utility decreases with increasing delay depends on the application for which the stream is used. The delay for the 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 Babcock *et al.* [4], Jin *et al.* [19] or Abadi *et al.* [1]. An output message carries the highest timestamp from all messages used to create it. When the final output message arrives at a sink, the difference between the current time and the timestamp is measured and this difference is the end-to-end delay.

A few types of QoS functions used are shown in table 2. Each of these types represents a different business scenario. For example, the linear function represents a scenario when the cost of the delay is proportional to the 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 when the marginal cost of the delay diminishes, the longer a message is delayed. In other words, the user's marginal benefit for messages delayed for a long time is negligible. The convex function represents the case when the marginal cost of the delay increases the longer a message is delayed. The sigmoid function is a continuous equivalent of a step function, which represents the case where delays up to some deadline x are not penalized much; however, if delays exceed the deadline, then the penalty becomes very high. In general, we assume that the QoS function can be any differentiable, non-decreasing function.

We assume that all messages must be processed. We can deal approximately with reordering schedulers that lose messages for the case where the utility is a concave or sigmoidal function of delay. A lost message is treated as a message with infinite delay. If messages can be reordered then a lost message is merely one that will be delivered at time infinity.

The goal of scheduling is to maximize the long-term average of the utilities of the messages processed by all the streams of the system. Messages arrive at the system, are processed, and have an end-to-end delay and a corresponding utility. The sum of the utilities of all the messages processed by the system divided by the time in which the messages were processed is the utility per unit time provided by the system. The goal is to maximize the average utility per unit time, *i.e.*

$$\frac{1}{|F|}\lim_{t\to\infty}\frac{\sum_{i=0}^{|F|}\sum_{t=0}^{N_t^i} q_i(d_t^i)}{N_t^i}$$

where N_t^i is the number of events received by the stream i before time t and F is the set of flows where the flows are indexed by i.

Table 1. System defintion

Parameters	Description
$s_i \in S$	Set of servers in a topology.
C	Connectivity matrix for a topology, where c_{ij} is a link capacity between servers s_i and s_j if such link exists.
$f_i \in F$	Set of computation flows, where each flow f_i is a directed acyclic graph with each node being an indivisible computation step, $\tilde{s}_{ki} \in f_i$.
$m(\tilde{s}_{ki}) = s_j$	A mapping function m that assigns each step of each flow to a server in a topology.

Next we identify some characteristics of the distributed controls problem. These characteristics help explain why the problem is difficult.

Firstly, the number of stream graphs may be large, and the number of nodes per stream graph may be substantial. Also, streams can flow in different directions: downstream for one stream can be upstream for another. For example, stream A can flow from server 1 to server 2 while stream B flows from server 2 to server 1.

Secondly, the queuing aspects complicate the control decisions. We present a simple example to highlight some of the queuing aspects of the problem. Consider a system with 3 queues and 2 streams. Stream A is first served by server 1, and then by server 2. Stream B is also first served by server 1, but then it is next served by server 3. Thus server 2 serves only stream A while server 3 serves only stream B. Suppose there is one job from stream A and one from stream B in server 1, and suppose there are 100 jobs from stream B in server 3 while server 2 is empty. If the costs for both streams increase linearly, and costs for stream B increase at a faster rate than costs for stream A, what is the best policy? One approach is to have server 1 give priority to stream B because it has higher cost; however, rapid completion of the job on server 1 may merely cause it to wait in the queue for server 3. By contrast, completing a stream A job on server 1 will allow this job to be processed immediately on server 2.

If, however, jobs take a long time on server 1, then by the time a stream B job is completed by server 1, the queue of jobs for server 3 may have been finished, allowing this job to be processed immediately. Thus the control policy is based on the likelihood of jobs that are finished upstream being processed without queuing delays downstream.

A reasonable control policy is to use controls based on the total number of jobs downstream. If there are a 1000 B jobs downstream and only 10 A jobs downstream, then completing an A job before a B job is more likely to allow the completed A job to be processed rather than merely have the B job enqueued downstream. Such a control policy is not effective for 3 reasons: (1) it doesn't take into account new jobs that may arrive in the future, (2) it doesn't deal with the distribution of job service times and (3), the optimal control policy depends on which downstream queues are large and which are small. In general, the problem is an infinite horizon, infinite continuous state Markov decision process. We don't formulate the problem as a decision process here but evaluate simpler versions of this problem.

Now, we present two approaches for QoS-based scheduling in a distributed stream processing environment. The first approach is based on a processor sharing scheme and uses a centralized optimization algorithm to reduce system delay (section 3). The second approach uses feedback control to provide local scheduling with information needed to properly adjust local flow priorities (section 4). These algorithms are preliminary efforts of understanding issues in distributed control of networks to maximize delay-dependent utilities.

3. Processor Sharing-based Scheduling

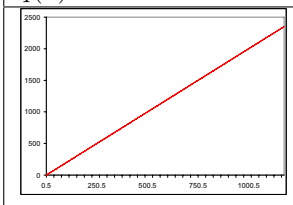
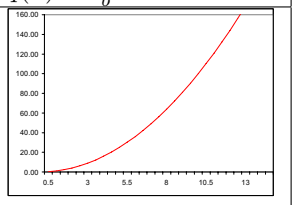
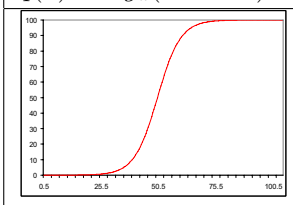
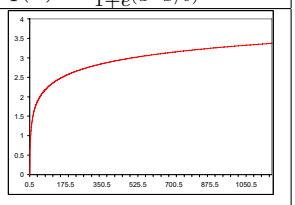
The first algorithm, we present here is based on the processor sharing scheme. In the literature, processor-sharing schedules give $1/k$ of a processor to each job when there are k jobs in the queue for that processor ($k > 1$). In this paper, by processor sharing we mean that all the streams share the processor according to weights assigned to stream; the proportion that each stream gets is independent of the number of jobs of that stream waiting for service. 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

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

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

The mathematical analysis of systems in which a fraction of a server is reserved for each stream is straightforward and yields closed-form solutions. By contrast, when a server's time is not assigned to an empty stream, the analysis does not yield closed-form solutions. Reserving a fraction of a server for a stream that is empty is not the optimal policy if preempting and resuming a job has no overhead. Therefore, we don't deal with the reservation case in this paper except to mention that bounds on how badly the reservation policy is can be calculated and are proportional to the number of streams.

Table 2. Types of QoS functions

Linear QoS	Convex QoS
$q(x) = a * x + b$	$q(x) = \frac{x^a}{b}$
	
Sigmoidal QoS	**Concave QoS**
$q(x) = log_a(b * x + c)$	$q(x) = \frac{w}{1+e^{(a-x/b)}}$
	

Given, this processor sharing algorithm, what are the values of $p_1, ..., p_n$ such that the expected cost for each flow is minimized? To determine these values, we represent the algorithm as a continuous-time Markov process. In this model we assume that the arrival processes for each stream is an independent Poisson process and that the service times are independent exponential random variables. 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.

The state of a server is an n-tuple, where n is the number of streams, where the k-th element of the tuple is the number of jobs from stream k waiting for service from that server. For every state there is a transition due to the arrival of one job from each stream. The rate of transitions due to the arrival of a job from stream k is λ_k. For every state with at least one job from stream k at the server, the rate at which the stream is served is ω_k, and hence the rate at which it leaves the server is $\omega_k \mu_k$ where μ_k is the rate at which it leaves the server if stream k is the only stream at this server.

Once the process is defined, we can find equilibrium probabilities ($\pi Q = 0$). From the equilibrium probabilities, the 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 the expected cost under given shares, $p_1, ..., p_n$ to a first approximation using the formula below.

$$\overline{Cost_i} = \sum_{i=0}^{c\overline{D}} re^{-ri} q_i(i) \text{ where } r = \frac{1}{\overline{D}}$$

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

It should be noted that the evaluation of the equilibrium probabilities is computationally intensive if done using the standard algorithms since the model has billions of states. For example, the number of states for the model of five queues whose queue size does not exceed ten is 10^5. And therefore the size of the Q-matrix is 10^{5^2}.

We have created a fast, small-memory approximation algorithm for the invariant probability evaluation. First, we note that we can convert the Q-matrix into the P-matrix containing 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 the P-matrix implicitly using the formula above. The invariant probabilities can be found by performing the iteration algorithm:

$$\pi_t P = \pi_{t+1}.$$

Moreover, π_t can be represented as a segment tree because most elements of π_t equal to zero and non-zero elements occur in consecutive subsequences. We further reduce the size of π_t by rounding down elements of π_t that are within ϵ of 0. The detailed evaluation of this algorithm together with the related proofs are too long to be presented in this paper and may be found in Khorlin [20]. Here we merely state, without proof, that the computational times are in the order of seconds on laptops when the numbers of streams and maximum queue sizes are in the single digits. (By maximum queue size we mean a size beyond which the queue grows with very small probability.) The accuracy of the model for maximum queue sizes that allow for rapid computation is shown in figure 1.

With the model that can predict the impact of the parameter values on our scheduling, we can create an optimization strategy to find the optimal parameters for each server in a topology. Since, the QoS function may not be convex, we use a generic optimization scheme: genetic algorithms [39]. When the QoS function is convex, standard hill-climbing solutions from the convex-programming literature can be used to find solutions with rapid computation times. When the QoS function is concave or sigmoidal, there can be multiple local optima. Therefore we use genetic algorithms in which the genome is a two dimensional array consisting of $p_1, ..., p_n$ for each server. And the genome evaluation function is our model. At the runtime, the system monitors the parameters at each server (*i.e.* arrival and service rates). When they change by more than ϵ, a re-optimization step 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.

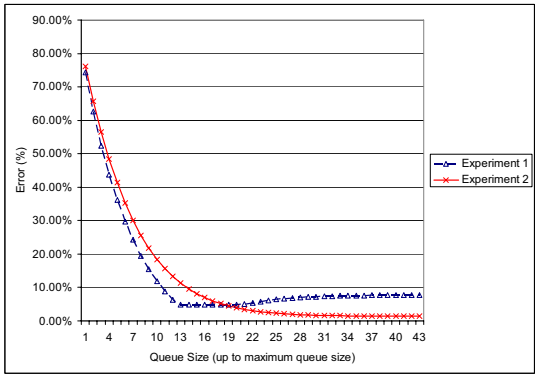


Figure 1. Markov chain model accuracy

4. Marginal Cost-based Scheduling

The processor sharing algorithm performs analysis based on the expected stream statistics. However, if the stream statistics change rapidly, such algorithms may not react fast enough. Thus, we present an alternative scheduling scheme that makes the scheduling decision at each Δt. For this algorithm, we assume that the cost of switching between the tasks is negligible.

In a single-server case, our scheduling algorithm works as follows. At each Δt, the algorithm inspects all unfinished messages for all local flows. The probability that a job in stream i will leave in the next Δt is $\mu_i \Delta t$ if stream i has the entire server to itself. The incremental cost of a message, as specified by the QoS function, is the slope of the function at time T, where T is the length of time the job has been in the queue. Thus the incremental benefit of serving this job is:

$$\mu_i \Delta t \left. \frac{dq_k(x)}{dx} \right|_{x=T_i}$$

where μ_i is the expected service time of the message. T_i is the amount of time this message spent in the system. For a message delivered to the end users, T_i, is the total delay used to compute the ultimate cost.

Then, for the next Δt, the algorithm schedules a message with the highest metric value. In a single server environment, we have shown that this algorithm minimizes the expected cost provided jobs can be preempted and resumed with no overhead cost. In essence, the metric computes the marginal cost of delaying a message for another Δt amount of time. Thus, scheduling the message with the highest cost minimizes the expected cost.

Our algorithm may result in messages being processed out of order. In fact, in a single-server environment, we have shown that if the function is convex, the scheduling policy degenerates into first come, first serve scheduling (FCFS) in which case, messages are processed in order. And if the function is concave, the scheduling policy is last come, first serve preemptive resume (LCFS-PR) in which case messages may be processed out of order [20]. In some stream processing systems, message reordering is not allowed [1]. When message reordering is not allowed, we modify our scheduling algorithm .

The modified algorithm only selects messages from the heads of the queues and uses a different selection metric. The idea is to select that stream for which completion of a job in the next incremental time interval has the maximum payoff. The metric is

$$\mu_i \Delta t(\gamma + \phi)$$

where γ represents the marginal cost of delay similar to the previous case, and

$$\gamma = m_c(T_i)$$

$$m_c(t') = \left. \frac{dq_i(x)}{dx} \right|_{x=t'} \Delta t$$

where ϕ 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.

In a multi-server environment, the scheduling algorithm is modified. Local schedulers must estimate delays downstream. To deal with these downstream delays, we provide a notion of the *feedback information* ω that is sent from downstream servers to upstream servers. This information flows in the opposite direction to the data messages and contains an estimate of the current delay on a server. When a server receives ω, it can adjust marginal cost computation as follows:

$$m_c(t') = \left. \frac{dq_i(x)}{dx} \right|_{x=(t'+\omega)} \Delta t.$$

This distributed scheduling approach can be viewed as a *feedback control system*. Currently, the feedback, we are using, consists of the cumulative expected service time for all downstream servers (*i.e.* single integer). This feedback does not take into account the queuing delays accurately.

Table 3. Experiment parameters

	Experiment 1		Experiment 2	
	Steam 1	Steam 2	Steam 1	Steam 2
λ^{-1}	250	250	190	250
μ^{-1}	100	100	100	70
$[p_1, p_2]$	1	10	4	7
q_{max}	44	11	33	13

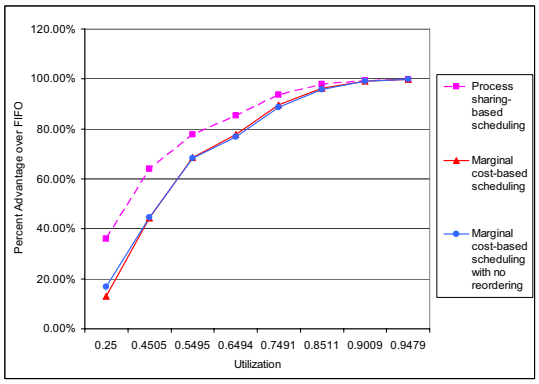


Figure 2. Scheduling in a single-server environment

Queuing delays depend on the rate with which the scheduling algorithm selects messages. This, 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 the local queues and the scheduling algorithm. In addition, queuing delays change rapidly. A meaningful way of providing feedback is by accumulating statistics in a sliding window. The size of this window impacts the scheduling and vice-versa, creating another feedback loop. All these issues complicate the development and analysis of the feedback algorithm.

Since service times are distributed exponentially, most true service times are below the expected service time. The expected service time metric overestimates the true service time, and thus takes into account some amount of the queuing delay. We have verified this hypothesis by making ω, the true future service for each messages. Such a change in the feedback did not result in better algorithm performance [20]. We are continuing to work on adding more precise information on queuing delays in feedback messages.

5. Experimental Results

The use of algorithms, such as first-come-first-served and traditional processor-sharing that do not take into account QoS functions can be shown to be arbitrarily bad compared with algorithms that take QoS functions into account. In these experimental studies, we next consider situations in which our algorithms do not perform exceptionally well because this sort of analysis is more useful than the analysis of situations in which QoS-based algorithms are guaranteed to perform arbitrarily better. Now we show how our algorithms perform in a single and multi-server environments. The Ptolemy discrete-event simulator is used for the performance evaluation [9]. We extended the Ptolemy runtime with a set of custom components that execute the scheduling algorithms we defined.

We showed that the Markov process outlined in section 3 accurately predicts the delays of our processor sharing algorithm. Two flows are located on one server parameterized as stated in table 3. The approximation is performed assuming a certain limits on the queue sizes for each of these two flows. The first set of experiments is performed on streams with equal arrival rates. The second set of experiments had streams with fast and slow arrival rates. The approximation error, as a percentage, is plotted in relation to the queue size. Both sets of experiments show that the approximation is reasonably accurate (lower than 10% error) for queue sizes greater than 12 (Figure 1).

In the second set of experiments, we ran our scheduling algorithms on a single server. The server has two streams with convex QoS functions. Stream 1 QoS is $2x^2$. And stream 2 QoS is $x^3/200$. The two QoS functions cross at $x = 400$; stream 1 has higher cost for low values of x and stream 2 has higher cost for high values of x. Convex QoS functions represent the case when the greater delays result in the 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 plots the percentage decrease in cost of the advantage of our algorithms over FIFO as a function of utilization. As utilization increases, the advantage of all three algorithms grows because the mean queue size increases. The processor sharing algorithm slightly outperforms marginal cost algorithms due to the fact that the selection of time interval, Δt, effects the accuracy of our scheduling. Δt cannot be made too small since it will result in inefficient execution. Moreover, the reordering does not 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 ran 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, $c * x + 100$ where $c \in [0, 25]$. Flow 1 has a sigmoidal QoS, $\frac{100000}{1+e^{15-x/10}}$. In each iteration of this experiment, we increased the slope of the linear QoS, c, in order to change

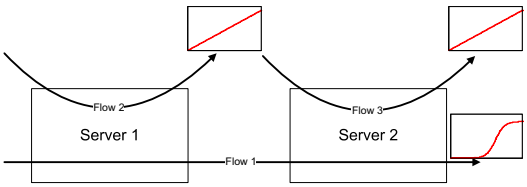


Figure 3. Experimental server topology

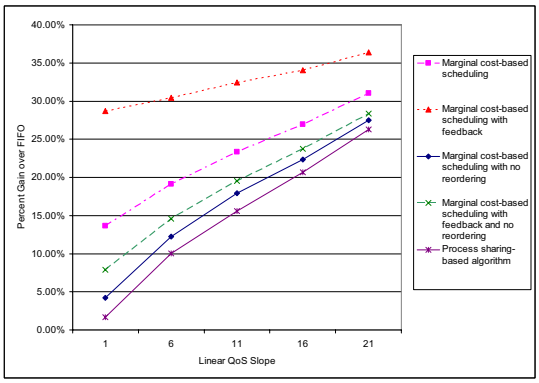


Figure 4. Scheduling in a multi-server environment

the trade-off between stream 1 and streams 2 and 3. In this round, we show that our algorithm with feedback performs better.

The results are shown on figure 4. Feedback improved the behavior of the marginal cost-based scheduling. Without the feedback the scheduling delays messages on the first server so long enough, that maximum cost is reached on the second server. With the feedback, the scheduling on server 1 is aware that more processing for flow 1 is still to come, and so it prioritizes flow 1 over flow 2. The processor sharing algorithm does not perform as well for two reasons. The way we compute the cost from the predicted average delay introduces an error and the genetic algorithm produces only near-optimal results. Secondly, scheduling with no reordering does not benefit from feedback, because with no reordering the queue dynamics play larger role. Since, the queuing delays are not captured in feedback messages, the efficacy is less compared to scheduling with reordering. Lack of reordering complicates the analysis beyond the issues outlined in section 4. A more extensive performance evaluation can be found in Khorlin [20].

6. Conclusion

Stream processing systems create new demands for system robustness, performance and adaptability. Concepts from control systems has been shown to be effective not only in the areas outside of computer science, but also for design of new networking algorithms [27]. We have presented preliminary analysis of simple models that explore some of the issues in distributed control of streams.

We presented two scheduling algorithms. One is an open-loop algorithm based on queuing theory. And the other is based on the distributed feedback control. These algorithms were used to explore tradeoffs between different parameters in distributed control.

7. Future Work

A great deal of work needs to be done on models that yield closed-form solutions or numerical solutions that can be computed rapidly. Initially, understanding the behavior of QoS based on delays under different control policies needs to be understood and explored with even simplistic models. Later, more sophisticated models that may not be amenable to closed-form or numerically-rapid algorithms should be developed and analyzed.

More importantly, we will combine distributed control approaches with algorithms that map stream-processing graphs on to hardware graphs [36]. Continuous adaptation using control policies and occasional remapping of stream graphs to hardware resource graphs form a coupled problem because each part influences the other.

8. Related Work

In general, the area of scheduling is very extensive, especially, in the domains of operating systems and networking [22, 24, 38]. Since our work is uniquely focused on distinct aspects of the distributed data-stream processing systems, we discuss only the work that is directly related to the scheduling in a stream processing applications. A good description of the requirements for such systems is provided in Bourbonnais *et al.* [8].

There are several stream processing systems that are being developed in the academia and industry. One of such systems is SMILE (Smart-Middleware Light-Ends) that is being developed at IBM Research [19]. SMILE introduces a novel correctness guarantee that results in the simplified operator semantics and the fault-tolerance algorithms. Also, as part of the 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 can be easily implemented in the SMILE system.

Borealis is one of the first stream processing system that used the 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 Abadi *et al.* [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 is based on the Aurora scheduling [10]. The Aurora scheduling consists of several schemes 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 Abadi *et al.* [1] and [10], since our algorithm tries to use auxiliary information that travels in messages to predict message impact of the scheduling policy on the 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 Abadi *et al.* [1]. Moreover, we also take into account the effect of queuing. As was shown, queuing can 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 will be interesting to compare different scheduling strategies and incorporate other types of QoS functions presented in Abadi *et al.* [1] into our framework and use the control theory to design a more robust system.

Event correlation engines are a special type of the stream processing systems that are very popular in the enterprise computing [2, 14]. Active Middleware technology (Amit) is one of leading correlation engines [2]. Amit defines a rich and extensive language and associated runtime for the event detection and correlation. In the future, for a correlation system like Amit, the distributed implementation of the runtime can 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 [21]. 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 [23]. 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 still has to be performed by the runtime 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 the query processing and minimize memory use. The scheduling algorithm proposed as part of STREAM focuses on the 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 the distributed environments, where the local choices may negatively effect the latency on the downstream server machines due to queuing delays.

There are many other stream processing or continious query systems being developed [11, 12, 13, 15, 16, 17, 28, 32, 34, 35]. Several of them are designed specifically for the network traffic monitoring domain [16, 34]. Some propose a general service infrastructure for development of the stream processing systems [17, 32]. Many of these systems are either not distributed or use shared memory architectures. However, some of them are distributed [17, 32], and can benefit from our scheduling algorithms.

Lastly, Δ-data flows represent a different and novel snapshot-based paradigm for the stream processing [25]. The scheduling algorithms presented here do not extend to the Δ-data flows environment. In Zimmerman *et al.* [40], several alternative algorithms for the scheduling of the 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 [18], Oracle Application Server [26], BEA ESB [6], SAP [30], and TIBCO ESB [37]. Our scheduling algorithms can 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 the steady-state approximation of Markov chains using aggregation-disaggregation [7, 33], and the approximate mean-value analysis [29] for estimating the expected queue sizes. However, the structure of our Q-matrix is not suitable for these approaches. Instead our approximation exploits the unique structure of the 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 can serve as an example of how we can apply the control theory to the scheduling in a stream processing system [27]. Generally, the area of stochastic control and its applications in system analysis are the tools that we will continue to use in our future work on the control-based, distributed scheduling algorithms [24, 31].

References

[1] D. J. Abadi, Y. Ahmad, M. Balazinska, M. C. Ugur Centintemel, J.-H. Hwang, W. Lindner, A. S. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Zdonik. The design of the borealis stream processing engine. In *CIDR*, 2005.

[2] A. Adi and O. Etzion. Amit - the situation manager. *VLDB J.*, 13(2):177–203, 2004.

[3] B. Babcock, S. Babu, M. Datar, R. Motwani, and D. Thomas. Operator scheduling in data stream systems. *VLDB Journal on Data Stream Processing*, 2004.

[4] B. Babcock, M. Data, R. Motwani, and J. Widom. Models and issues in data stream systems. In *ACM Symp. on Principles of Database Systems*, 2002.

[5] S. Babu and J. Widom. Continuous queries over data streams. In *SIGMOD Record*, September 2001.

[6] BEA Corp. BEA aqualogic service bus. http://www.bea.com, 2006.

[7] G. Bolch, S. Greiner, H. de Meer, and K. S. Trivedi. *Queueing Networks and Markov Chains*. John Wiley & Sons, Inc., 1998.

[8] S. Bourbonnais, V. M. Gogate, L. M. Haas, R. W. Horman, S. Malaika, I. Naran, and V. Raman. Towards an information infrastructure for the grid. *IBM Systems Journal*, 43(4):665–688, 2004.

[9] J. T. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt. Ptolemy: A framework for simulating and prototyping heterogeneous systems. *Int. Journal of Computer Simulation*, 4:155–182, April 1994. Special issue on "Simulation Software Development".

[10] D. Carney, U. Cetintemel, A. Rasin, S. Zdonik, M. Cherniack, and M. Stonebraker. Operator scheduling in a data stream manager. In *Proceedings of the 2003 International Conference on Very Large Data Bases*, September 2003.

[11] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthya, S. R. Madden, V. Raman, F. Reiss, and M. A. Shah. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *CIDR*, 2003.

[12] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. Niagaracq: A scalable continuous query system for internet databases. In *SIGMOD*, 2000.

[13] L. Chen, K. Reddy, and G. Agrawal. Gates: A grid-based middleware for processing distributed data streams. In *Thirteenth IEEE International Symposium on HighPerformance Distributed Computing (HPDC-13)*, June 2004.

[14] C. Corp. Corel8. www.corel8.com, 2006.

[15] C. Cortes, K. Fisher, D. Pregibon, A. Rogers, and F. Smith. Hancock: A language for extracting signatures from data streams. In *Proceedings of the 6th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 9–17, 2000.

[16] C. Cranor, T. Johnson, and O. Spatscheck. Gigascope: a stream database for network applications. In *Proceedings of ACM SIGMOD*, June 2003.

[17] P. B. Gibbons, B. Karp, Y. Ke, S. Nath, and S. Seshan. IrisNet: An architecture for a world-wide sensorweb. In *IEEE Pervasive Computing*, pages 22–33, October 2003.

[18] IBM Corp. Websphere enterprise service bus. http://www.ibm.com, 2006.

[19] Y. Jin and R. Strom. Relational subscription middleware for internet-scale. In *Proceedings of the 2nd International Workshop on Distributed Event-Based Systems (DEBS'03)*, 2003.

[20] A. Khorlin. Scheduling in distributed stream processing systems. Master's thesis, California Institute of Technology, 2006.

[21] R. Koster, A. Black, J. Huang, J. Walpole, and C. Pu. Infopipes for composing distributed information flows. In *Proceedings of the 2001 International Workshop on Multimedia Middleware*, 2001.

[22] R. Krishnamurthy, S. Yalamanchili, K. Schwan, and R. West. Share-streams: A scalable architecture and hardware support for high-speed QoS packet schedulers. In *Symposium on Field Programmable Custom Computing Machines (FCCM) IEEE*, April 2004.

[23] V. Kumar, B. F. Cooper, and K. Schwan. Distributed stream management using utility-driven self-adaptive middleware. In *IEEE International Conference on Autonomic Computing*, 2005.

[24] B. Li and K. Nahrstedt. A control-based middleware framework for quality of service adaptations. In *IEEE J. Selected Areas of Communication*, 1999.

[25] R. Manohar and K. M. Chandy. Δ-dataflow networks for event stream processing. In *16th IASTED International Conference on Parallel and Distributed Computing and Systems*, November 2004.

[26] Oracle Corp. Oracle enterprise service bus. http://www.oracle.com, 2006.

[27] F. Paganini, Z. Wang, J. C. Doyle, and S. H. Low. Congestion control for high performance, stability and fairness in general networks. In *IEEE/ACM Transactions on Networking*, pages 43–56, February 2005.

[28] D. S. Parker, R. R. Muntz, and H. L. Chau. The tangram stream query processing system. In *Proceedings of the 1989 International Conference on Data Engineering*, pages 556–563, February 1989.

[29] D. C. Petriu and C. Woodside. *Approximate mean value analysis based on Markov chain aggregation by composition*. Linear Algebra and its Applications. Elsevier Science Journal, 2004.

[30] SAP. SAP. www.sap.com, 2006.

[31] L. I. Sennot. *Stochastic Dynamic Programming and the Control of Queueing Systems*. Probability and Statistics. Willey, 1999.

[32] J. Shneidman, P. Pietzuch, J. Ledlie, M. Roussopoulos, M. Seltzer, and M. Welsh. Hourglass: An infrastructure for connecting sensor networks and applications. Technical report, Harvard Technical Report TR-21-04, 2004.

[33] J. C. Strelen. Approximate product form solutions for markov chains. *Performance Evaluation*, 30(1–2):87–110, 1997.

[34] M. Sullivan and A. Heybey. Tribeca: a system for managing large databases of network traffic. In *Proceedings of the USENIX annual technical conference*, pages 15–19, June 1998.

[35] D. Terry, D. Goldberg, D. Nichols, and B. Oki. Continuous queries over append-only databases. In *Proceedings of the 1992 ACM SIGMOD international conference on Management of data*, pages 321–330, San Diego, California, United States, 1992. ACM.

[36] L. Tian. Resource allocation in streaming environments. Master's thesis, California Institute of Technology, 2006.

[37] TIBCO. TIBCO enterprise service bus. http://www.tibco.com, 2006.

[38] R. West, Y. Zhang, K. Schwan, and C. Poellabauer. Dynamic window-constrained scheduling of real-time streams in media servers. *IEEE Transactions on Computers*, June 2004.

[39] D. Whitley. A genetic algorithm tutorial. *Statistics and Computing*, 4:65–85, 1994.

[40] D. M. Zimmerman and K. M. Chandy. A parallel algorithm for correlating event streams. In *19th IEEE International Parallel and Distributed Programming Symposium (IPDPS 2005)*, April 2005.

Proceedings of the IEEE Services Computing Workshops (SCW'06)
0-7695-2681-0/06 $20.00 © 2006 IEEE