

Utility-aware Resource Allocation in an Event Processing System

Sumeer Bhola, Mark Astley, Robert Saccone and Michael Ward
{sbhola, mastley, rsaccone, mjwmjw}@us.ibm.com
IBM T.J. Watson Research Center
19 Skyline Drive
Hawthorne, NY 10532

Abstract—

Event processing systems link event producers and consumers in a flexible manner, by supporting multiple communication patterns and powerful message transformations. Such systems support both loosely coupled communication patterns, such as one-to-one or one-to-many, and tightly coupled (request-response) patterns. An important concern in these systems is the allocation of resources used to support event processing. These include traditional constrained resources such as CPU and network, as well as an increasing reliance on storage resources. For instance, transactional and reliable producers require events to be logged to stable storage.

In this paper we consider the problem of utility driven allocation of these resources, taking into account resource capacity. In contrast to previous work, we (1) develop a unified utility model to deal with throughput requirements of one-way flows and latency requirements of request-response flows, (2) differentiate between classes of consumers for one-to-many flows and subject these classes to consumer admission control, (3) compute system optimization constraints using both resource bandwidth and latency, and (4) develop a control architecture that combines periodic optimization with continuous rate control, both of which are utility aware. We describe an experimental evaluation of our approach on a prototype event processing server.

I. INTRODUCTION

Event processing systems link event producers and consumers in a flexible and powerful manner, by supporting multiple communication patterns and enabling transformation and filtering of messages as they flow from producers to consumers. The class of event processing systems includes (1) messaging middleware [1], (2) event *stream processing* systems [2], [3], and (3) systems supporting the Enterprise Service Bus [4] architecture for service-oriented integration. Communication patterns include both loosely-coupled (one-way), and tightly-coupled (request-response) interactions. Similarly, transformation and filtering may be accomplished directly by system-hosted application code, or implicitly using declarative specifications as in content-based publish/subscribe [1].

Event processing systems provide these services by allocating CPU, network and storage resources on behalf of users. For instance, persistent storage may be allocated to support transactional producers. However, since system resources are bounded, and since applications may have widely varying requirements and importance, a significant problem is the optimal allocation of resources. We illustrate the problem with a few simple workload examples:

- *Trade Data*: An application is producing events corresponding to each trade in the stock market for a certain market segment. There are two kinds of consumers interested in this data: (1) consumers at one or more brokerage firms, called *gold* consumers, which pay for the data, and (2) *public* consumers connected over the Internet. The *gold* consumers have a higher priority because they bring more benefit to the system. A consumer attaches to a node in the system, and receives the events it is interested in. Before being provided to public consumers, events are altered within the system and certain fields available only to gold consumers are removed. In addition, gold consumers expect reliable delivery, which places extra overhead on the system to process acknowledgments. The event flow needs to be delivered with low latency, and therefore, is not very *elastic* [5] in terms of rate, i.e., the rate cannot be decreased to tolerate delays. In case of lack of resources, the system can reduce load by using admission control to deny service to public consumers.
- *Latest Price Data*: An application is producing events representing the latest prices of IBM stock. Public consumers connected to the system receive price events which satisfy a consumer-specified filter (e.g., price > 80). That is, for each price event, the system evaluates the filter to determine whether the event should be delivered to the consumer. The event flow is very elastic, since rate can be decreased (and latency increased) by reducing the frequency of updates. In case of lack of resources, the system can reduce load by either reducing the producer rate, denying service to consumers or both.

In the above scenarios, system resources are consumed both on a per event basis, independent of the number of consumers, and on a per event, per consumer basis. The cost of the latter can vary depending on the complexity of the per consumer processing, like content filtering, reliable delivery, etc.

In this paper, we consider the problem of allocating resources in an event processing system such that user “utility” is maximized. Utility is an abstract measure of benefit to the user which we seek to maximize given the available resources. Our approach consists of a utility model which expresses user benefit in terms of allocated rate, allowable latency, and number of consumers; a resource model which defines the constraints

of the available resources; an optimization algorithm which is executed periodically to maximize utility within the constraints of the resource model; and, a runtime system which enforces resource allocations, implements continuous rate control, and provides feedback into the optimization mechanism.

A novel feature of our approach is the unification of utility for both one-way and request-response flows into a single model. That is, while previous work has typically considered utility models in isolation, such as latency-based models for request-response interactions in web servers, or rate-based models for bandwidth allocation in communication networks, we have developed an approach to transform round-trip latency requirements into rate requirements in order to simplify optimization. Likewise, our utility model must also consider consumer admission, since the set of potential consumers may scale beyond system capabilities. Our utility model assigns a utility function to classes of consumers, and incorporates population size into the utility computation. In contrast, related work on resource allocation in event processing systems does not consider consumer admission control.

We have developed a resource model using a blend of online and offline modeling techniques. The model constrains the allocation of resources according to both resource bandwidth and the latency incurred by using a resource. In particular, bandwidth places aggregate constraints on multiple users of a resource, while the latency incurred in using a resource places individual constraints on each user. Given the fine-grained multiplexing of message flows within a server, it is hard to estimate resource bandwidth models online. Therefore, we construct offline bandwidth models by fitting parameterized equations to carefully controlled test scenarios, and use online measurement for error correction. Conversely, resource latency is sensitive to overall load on a server, making it difficult to develop offline models. In this case, we construct models online by instrumenting processing paths.

The utility and resource models are used by a runtime resource manager which combines periodic optimization and utility aware continuous rate control to manage resources. The optimization problem does not lend itself to standard techniques in related work due to the non-concavity of the objective function and non-convexity of the constraints. We describe an efficient greedy optimization approach based on the benefit-cost ratio of optimization variables. Similarly, we derive a utility aware rate control mechanism, inspired by network flow control [6], [7], which is used to adjust rate values between optimization events and prevent congestion collapse.

Note that in contrast to related work, we require both optimization and utility aware rate control. For instance, web server resource allocation has typically only used periodic optimization (say with a period of 30 seconds) and/or a scheduling algorithm. For web workloads, rate control is not necessary because of (1) the inherent throttling built into request-response loads which prevent significant overload due to changes between optimization events, (2) the single entry point to the system is controlled by the scheduler which

is aware of system utilization. An event processing system may scale to a wide-area environment with multiple entry points, and supports both one-way and request-response interactions; therefore, rate control can not be ignored. Conversely, techniques for communication networks have focused on optimization *flow control* [6], which uses near-continuous rate optimization (say with a period of 1 second) to adjust rate allocations and prevent congestion collapse. Flow control alone is sufficient in this case because only rates are optimized, not consumer population.

The remainder of the paper is organized as follows. Section II describes the utility model and Section III the resource modeling approach. Section IV describe the control architecture and components. Section V presents a detailed experimental evaluation. Section VI describes related work and we conclude in Section VII.

II. UNIFIED UTILITY MODEL

We model user requirements in terms of utility, which reflects the degree to which user requirements have been met. The system uses utility to determine resource allocations which maximize total utility. Utility is described using two logical abstractions, flow and consumer class:

Flow: A flow is used to group related messages that have similar quality requirements. For instance, all stock quote messages for a particular market segment may be grouped together as a single flow. We consider two types of flows: (1) *one-way*, and (2) *request-response*. One-way flows are further classified into two types, *one-to-one* and *one-to-many*. Content-based publish/subscribe is an example of a one-to-many flow, where a single message may be delivered to multiple recipients. Similarly, message queues in middleware applications are examples of one-to-one flows. Note that messages in a one-to-one flow are delivered to exactly one consumer, even when there are many potential consumers for each message.

Consumer Class: For a one-to-many flow, a consumer class groups flow consumers that have the same importance and resource requirements. For instance, high-premium consumers may be grouped into a 'gold' consumer class. Each consumer class is associated with a particular one-to-many flow, and each flow may have multiple consumer classes. Consumer classes are only needed for one-to-many flows since a single message may be delivered to diverse sets of consumers. However, for one-to-one and request-response flows, the 'premium' of the single consumer is incorporated directly into the overall utility as described below.

Our goal is to express utility in terms of (1) the rate, r_{alloc} , allocated to a flow when it enters the system; and (2) for one-to-many flows, the number of consumers, n_{alloc} , admitted to the system for each consumer class. These parameters allow the system to allocate resources by exercising control at the endpoints of the system, i.e., enforcing r_{alloc} at the entry points of messages, and n_{alloc} at the exit points of messages.

We now discuss how the utility functions are specified for the different types of flows.

A. Utility for One-Way Flows

For a one-to-one flow i , utility, U_i , is expressed directly as a function of $r_{alloc}(i)$, e.g., $U_i(r_{alloc}(i))$ ¹. This function implicitly encodes the maximum rate at which message producers for this flow want to send messages, say r_{max} . In particular, $U_i(r_{alloc})$ will not increase for $r_{alloc} > r_{max}$. Additionally, a value r_{min} is specified as a lower-bound on the rate allocation. In this paper we assume that U_i is non-decreasing, continuous, and concave for $r_{alloc} > r_{min}$. Our experiments use piecewise linear utility functions.

For one-to-many flows, utility is expressed per consumer. All consumers in a class have the same utility function, and the utility of the class is the sum of the utility of each consumer in that class that is receiving service. Therefore, the utility of a class j corresponding to flow i , is $n_{alloc}(j)U_j(r_{alloc}(i))$. The properties of U_j are the same as described previously for one-to-one flows. The total utility of the flow is the sum of the utilities of each class for the flow.

Although we have expressed each consumer's utility function in terms of r_{alloc} , consumers may observe a different rate due to in-network filtering or aggregation. However, because this observed rate will be a function of r_{alloc} , we believe it is acceptable to express consumer utility in terms of the rate at which the flow entered the system.

Moreover, whereas previous work [8], [9] has focused on end-to-end latency and ignored flexibility in rate, we instead focus solely on rate. We believe flexibility in rate is a key mechanism for controlling resource allocation, and is an important factor in determining end-to-end latency since lowering event rate increases the end-to-end latency of information propagation. In our experience, propagation delays due to processing latency are typically insignificant (on the order of 100ms) compared to the latency requirements of most consumers of one-way flows. Other previous work [10] has considered utility in terms of raw network bandwidth and not in terms of a user metric like flow rate, and additionally ignored other resources like CPU and storage.

B. Utility for Request-Response Flows

Utility is expressed as a function of round-trip latency, T_{rt} , averaged over a number of requests. Our objective is to translate latency-based utility into utility in terms of r_{alloc} . Informally, if function A : latency \rightarrow utility maps latency to utility, then we seek a function B : rate \rightarrow latency so that $A(B(r))$: rate \rightarrow utility where r is a rate. We derive B by determining the expected round-trip latency for various values of r_{alloc} as follows.

For this paper, we assume the request handlers, i.e., responders, are application code hosted by the system and therefore consume system resources. We compute an online estimate of the number of request senders, N , and their average think time,

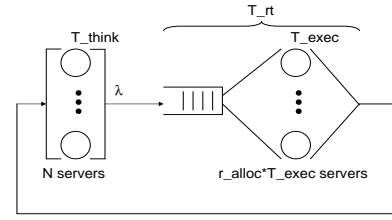


Fig. 1. Queuing network model to transform r_{alloc} to latency.

T_{think} . Assuming an uncongested system, e.g., the system maintains itself in an uncongested state using our control techniques, the average time interval between admitting a request into the system and sending back a response to the requester can be estimated online. We denote this estimate as T_{exec} , the 'execution' time of the request. Execution time depends on the utilization of resources in the system, and can vary significantly, as the utilization changes. As resource utilization is mainly a function of workload and not of our resource allocation method (which partitions the utilization amongst competing users based on utility), T_{exec} is not significantly influenced by the resource allocation result, and is an acceptable input for determining rate allocations. Note that 'execution' time views the system as a black box. For example, the system could be deployed over a wide-area network which routes the request over multiple hops. However, these details are abstracted in the value T_{exec} . In addition to T_{exec} , requests will experience a queuing delay which depends on the rate allocation to the flow at the entry point to the system.

We combine N , T_{think} and T_{exec} to estimate mean round-trip delay, T_{rt} , using mean value analysis (MVA) of a queuing network model [11]. Figure 1 shows the queuing network model. The number of 'servers' in the model is the number of requests that can be concurrently 'executing' in the system, and corresponds to the number of requests admitted in the time interval after the admittance of a particular request r and before the response is sent for r . Since the length of this interval is T_{exec} and the unit rate of admittance is r_{alloc} , the number of requests admitted in this interval is $r_{alloc} \times T_{exec}$. We can compute latency by solving this model for values of r_{alloc} such that $r_{alloc} \times T_{exec}$ is an integer (since the number of 'servers' should not be fractional). The intermediate values of T_{rt} are estimated using linear interpolation.

Consider an example flow with $N = 20$, $T_{think} = 100ms$, $T_{exec} = 8ms$. We use MVA to compute the value of T_{rt} for 1, 2 and 3 'servers', which corresponds to r_{alloc} values of 125, 250 and 375 msg/s . These are the three data points displayed for the solid line in figure 2. Linear interpolation is used to predict the latency at other r_{alloc} values.

This approach to predicting latency can be inaccurate for multiple reasons. Since the lowest r_{alloc} value for which we have an MVA prediction is 125 msg/s , we can be arbitrarily inaccurate for r_{alloc} values less than this. Note that in the figure, this inaccuracy tends to underpredict latency. In the extreme case of a rate allocation of 0, the latency is incorrectly predicted to be finite. In addition, since our MVA model

¹We omit the flow labels on the rate notation, like r_{alloc} , and consumer class label from n_{alloc} , when it is clear from the context.

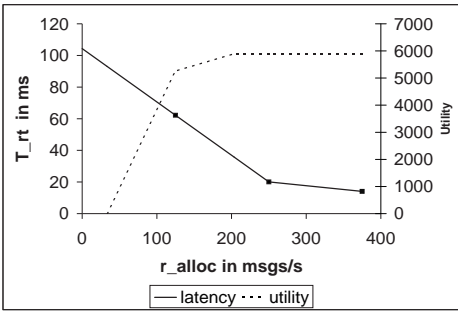


Fig. 2. T_{rt} , Utility as a function of r_{alloc} ($r_{min} = 33.3$, $r_{max} = 203.7$)

makes certain assumptions, like exponentially distributed inter-arrival times and service times, it is not accurate when those assumptions are violated. Inaccuracy in latency prediction causes incorrect prediction of utility for a particular rate allocation.

For our workloads, online testing revealed that our MVA model was overestimating T_{rt} for a given r_{alloc} , for high r_{alloc} values. To overcome this problem, we use a heuristic in setting r_{max} : the value of r_{max} is 10% more than the peak rate that can be achieved, i.e., $N/(T_{exec} + T_{think})$. To reduce the inaccuracy of interpolation at low r_{alloc} values, we set r_{min} to a positive value in a manner described next, and set utility at a rate allocation of r_{min} equal to 0. As future work, we intend to use an online measurement based approach to correct the function B : rate \rightarrow latency, for each flow, which would adapt to the unique inter-arrival time and service time characteristics of each flow.

As a concrete example of utility transformation, let utility as a function of latency be of the form $C(1 - T_{rt}/D)$, i.e., utility decreases linearly with increasing T_{rt} , where C, D are constants for a particular flow. The value D is a latency threshold that the system should not exceed, and therefore we set r_{min} equal to $N/(T_{think} + D)$ and utility equal to 0. At the value of r_{max} , we set utility equal to $C(1 - T_{exec}/D)$, i.e., there is no queuing delay. For r_{alloc} values between r_{min} and r_{max} , we use the T_{rt} estimate from MVA to compute the utility. Figure 2 shows the result of these utility computations for the same example flow, with $D = 500ms$, and $C = 6000$.

III. RESOURCE MODELS

Flows and consumers can utilize CPU, storage and network resources. In this section we discuss our approach to modeling the bandwidth of these resources. For storage, we also consider latency effects since this constrains the rate of certain flows.

A. CPU Bandwidth Modeling

We use a measurement based modeling approach to predict the CPU utilization for each server. For each one-to-one or request-response flow, i , we construct a model of the form $A_i \times r_{alloc}(i)$ which predicts CPU utilization at a given rate. A_i is a constant for the flow computed using a least-squares fit of a series of measurements. For a one-to-many flow, we construct a model of the form $A_i \times r_{alloc}(i) + \sum_j B_j \times r_{alloc}(i) \times n_{alloc}(j)$ where A_i is a constant for the flow, and

B_j are constants for the consumer classes. Both constants are computed using a least-squares fit of test measurements. The fine-grained multiplexing of flows in a server makes it difficult to isolate the effects of individual flows and consumer classes on overall CPU utilization. Thus, the constants for our flows are fitted offline on test measurements with a single flow and consumer class.

Our experiences with measuring and fitting a variety of flows has lead to the following observations:

- 1) For one-to-many flows, computing accurate values for A_i, B_j requires a large number of tests at different points in the workload space. For instance, we needed tests with a high rate and few consumers, and tests with a low rate and a large number of consumers. In some cases, fitting only using tests with high rate caused B_j to be six times the actual value, and only using tests with low rate caused the A_i value in the least-squares fit to be negative.
- 2) For reliable flows, that use both CPU and storage resources, the error (constant) term in the fit is high, leading to a less accurate model. For example, in the model we evaluate in Section V, the error term for reliable flows is three times larger than that for best-effort flows.
- 3) We have implicitly assumed that the individual models of each flow can be added to predict the total CPU utilization at a server with multiple flows. This assumption seems to work well in practice.

We believe a measurement based approach is promising, since it avoids developing detailed models of the software and hardware. In a running system, we use proportional error correction that equally scales the A_i and B_j coefficients when the predicted CPU utilization differs from the actual measured value. We discuss error correction in more detail in Section V.

B. Storage and Network Bandwidth Modeling

Storage and network resources constrain the rate at which messages are stored or routed, respectively, and are sensitive to both rate and message size. The storage system is only used for reliable flows. Each reliable message received from a producer at the entry point to the system is logged to persistent storage. Once the message is logged, the producer may need to be acknowledged, so that the producer transaction can commit.

Like the CPU model, for storage and network resources we constrain the aggregate rate of all flows. For storage, the aggregate rate is constrained by the peak message logger bandwidth in bytes/second. The peak message logger bandwidth is initially set to a value estimated offline. This is corrected using online measurements of queuing delay versus observed logger bandwidth. Likewise, for the network, we initially constrain according to the peak network interface bandwidth in bytes/second.

The other aspect of the resource model is computing the resource coefficients, A_i . Unlike the CPU model, where we estimated A_i offline, the constants in the storage and network

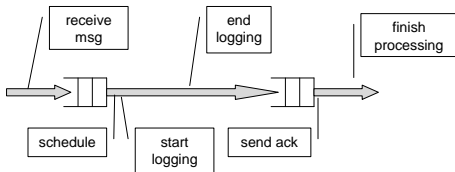


Fig. 3. Time line of latency incurred by a reliable message

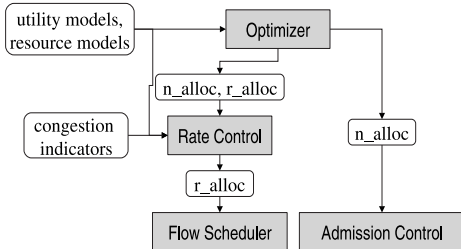


Fig. 4. Control System Architecture

models are estimated online by measuring the average size of messages in a flow.

C. Storage Latency Modeling

Latency is important for producers of reliable flows, since they need an acknowledgment before producing the next message, which places a constraint on the rate of each reliable flow. Consider a producer, that produces one message per transaction (this discussion can be generalized to multiple messages per transaction). Figure 3 shows the time line of message processing in the server from left to right. The message is first queued before scheduling (which enforces the rate allocation r_{alloc}), and then again after logging is complete. The producer will not produce another message until it receives the acknowledgment (ack) that the message has been logged. The latency incurred between the time the message is received to the time the acknowledgment is sent, minus the time spent in the scheduler queue, is the inherent latency encountered by this producer given the current system workload and is independent of the rate allocation given by the optimizer. This inherent latency places a bound on the rate at which the flow can produce messages and is continuously estimated in a running system. Without this bound on flow rate, the optimizer may allocate a rate that is higher than what can be achieved, and unnecessarily penalize other flows or consumers by not admitting them.

IV. ONLINE CONTROL

In this section we discuss the components that exercise control on the system. We begin with a description of the control architecture.

A. Architecture

The control system has four components: (1) optimizer, (2) rate control, (3) admission control, and (4) flow scheduler. The interaction between these components is depicted in figure 4. Admission control and the flow scheduler simply enforce the n_{alloc} and r_{alloc} values, respectively.

The optimizer runs periodically, say every 30 seconds, to compute n_{alloc} and r_{alloc} values for all the consumer classes and flows. In a distributed system, the optimizer may itself be distributed, and may also choose paths for flows through the system, and load balance consumers across multiple nodes in the system.

Rate control receives the n_{alloc} , r_{alloc} values from the optimizer, the utility and resource models, and may modify the r_{alloc} values based on congestion information in the system. That is, it may reduce some r_{alloc} values if resources are congested or increase values if spare capacity is available. Unlike the optimizer, rate control makes frequent changes to the rates, say once every second, so we can think of it as running continuously. Making rate control utility aware ensures that fluctuations in the system at a small time-scale are handled in a manner consistent with utilities.

B. Optimizer

The optimizer has the objective to maximize the total utility under both individual flow rate constraints, and aggregate constraints. The problem is challenging because the objective function is non-concave and the aggregate constraints are non-convex. Specifically, the objective function has terms of the form $n_{alloc}(j) \times U_j(r_{alloc}(i))$, which is non-concave even though U_j is concave. The aggregate CPU and network constraints define non-convex sets because of terms of the form $B_j \times r_{alloc}(i) \times n_{alloc}(j)$.

In [12], we describe a distributed optimization algorithm that computes n_{alloc} and r_{alloc} values for an overlay network of servers with multiple bottleneck resources. In this paper we consider the optimizer for a single server, since that is the context in which the evaluation is presented in the next section. The optimizer considers a single bottleneck resource, which is CPU bandwidth in our experimental evaluation.

We have developed a greedy algorithm that starts with all r_{alloc} values equal to r_{min} and all n_{alloc} values equal to 0, and increases the parameter with the highest benefit-cost ratio. We define the benefit of increasing a variable as the increase in the objective function if the variable is increased by Δ . Similarly, we define the cost of increasing a variable as the increase in the bottleneck resource consumption if the variable is increased by Δ . By relying on the following properties, (1) the benefit-cost ratio of increasing n_{alloc} is constant, i.e., does not depend on n_{alloc} , and (2) each utility function is concave, we can prove that the algorithm has time complexity $O(n \times m + m)$, where n is the number of flows and m is the number of consumer classes. In addition, the algorithm is fair when multiple variables have identical benefit-cost ratios.

Property (1) holds because of the form of our utility and CPU model functions. For example, the partial derivatives (with respect to n_{alloc}) of the utility and cost of one-to-many flows are $U_j(r_{alloc}(i))$ and $\sum_j B_j \times r_{alloc}(i)$, respectively. Property (2) holds by our assumptions in Section II.

This algorithm can be extended for multiple bottleneck resources in a single server context, such as CPU and storage bandwidth, in the following manner: run the above algorithm

multiple times by considering each resource as the primary bottleneck, and choose the run with the best final utility value. The primary bottleneck is used to define the resource cost in the benefit-cost ratio, thus controlling the order in which the greedy algorithm increases allocations, while respecting the constraints of all the bottleneck resources (not just the primary).

C. Rate Control

The rate control component attempts to continually maximize utility by adjusting r_{alloc} values assuming constant n_{alloc} . Rates are adjusted either when spare capacity is available (rates are increased), or when the system is congested (rates are decreased). Since n_{alloc} is assumed constant, the objective function for rate control is concave, and the constraints are convex.

The rate control approach we use is a generalization of a network flow control algorithm defined in [7] that can handle concave and piece-wise linear utility functions. Our generalization of this algorithm allows it to be used for heterogeneous resources, not just network bandwidth, and when utilities are not directly expressed in terms of resource usage.

Each resource maintains a boolean variable, where 1 indicates the resource is congested and 0 indicates uncongested. A resource periodically sends the value of this congestion indicator to the entry points of all flows that use that resource. At each flow entry point, the congestion indicators of the resources used are aggregated to periodically adjust the flow rate allocation. This occurs at each entry point independent of others, hence the algorithm is distributed.

The flow rate allocation is adjusted in the following manner. If none of the resources used by flow i are congested, the rate is increased additively by a value proportional to the partial derivative of the optimizer objective function with respect to $r_{alloc}(i)$. If some of the resources are congested, the rate is decreased additively by a value proportional to the sum of the coefficients of $r_{alloc}(i)$ in the resource constraint equations, summed across all the congested resources. The proof that this approach maximizes the objective function (utility) is similar to the proof in [7] and is omitted.

At each optimizer cycle, rates are “snapped” back to the values allocated by the optimizer. This allows rates to quickly adjust to re-optimized values and is desirable in the more typical case when the change in capacity is intermittent. If congestion or excess capacity occurs long term, then rates will periodically “snap” as illustrated in Section V.

V. EXPERIMENTAL EVALUATION

We evaluate our modeling and control algorithms in the context of a single-server prototype based on the Gryphon [13] publish-subscribe messaging system. Gryphon is a robust, highly-scalable messaging system that implements the publish-subscribe portion of JMS using both best-effort and reliable message delivery. Gryphon is representative of a large class of event processing infrastructures and consists of an overlay network of messaging *servers*, each of which may host

one or more network-connected *clients*, i.e., producers and consumers. We have enhanced Gryphon to host application code for processing requests and generating replies for request-response flows.

We define flows in terms of JMS topics, where each flow defines one or more classes as described in Section II. Parameters for the resource models described in Section III are either determined from isolated offline tests of Gryphon, or from runtime components which provide feedback to the runtime. The runtime components described in Section IV are implemented as Gryphon components which optimize rate and population allocation, perform admission control, enforce flow rates, and exert rate control.

We constructed several test scenarios in order to explore three aspects of our prototype system: 1) the ability of the system to make flow rate and population trade offs in a variety of circumstances; 2) the ability of the system to adapt rates to excess capacity; and 3) the effect of latency constraints when mixing distribution patterns such as one-way and request-response flows. We describe each category of experiments in the sub-sections below.

We conducted our experiments on a Gryphon server treating CPU as a bottleneck resource. The CPU resource available to the system is indicated by a configurable utilization threshold. The Gryphon server and clients were tested on a dedicated gigabit LAN consisting of several 6-way 500 MHz PowerPC servers each with 4 GB of memory running AIX. The server was executed on a dedicated machine. The clients were spread among the remaining machines so that only server resources were constrained.

Our test scenarios used best-effort and reliable one-to-many flows, and best-effort request-response flows. The constants for the resource models for the various flows were determined experimentally as described in Section III. For best-effort one-to-many flows, $A_i = 0.00146092$ and $B_j = 0.00103146$; for reliable one-to-many flows, $A_i = 0.0148354$ and $B_j = 0.00113717$; and finally, for request-response flows, $A_i = 0.1347943$. In all experiments, the optimizer ran every 20 seconds, and rate control ran every second.

A. Optimization Tradeoffs

The optimizer maximizes utility by making flow rate and population trade offs. These trade offs are affected by CPU capacity; and the per-flow CPU load, rate ranges, available populations, and utility functions.

We designed two experiments to evaluate optimization trade offs. The first experiment simulates a “trading day” of a stock trading event processing system and demonstrates dynamic optimization decisions that must be made as consumer populations change over time. The second experiment combines one-way flows with different reliability requirements. This experiment demonstrates optimization trade offs when flows have different impact on underlying resources.

In the “trading day” experiment, events describe the latest prices of stocks, i.e., latest matched orders. Flows in the system correspond to “sectors”, which group events for related stocks.

Segment	Gold	Silver	Bronze
Start of Day	100	100	100
Coffee Break	10	25	100
After Coffee Break	100	100	100
Lunch	0	5	100
After Lunch	100	100	100
End of Day	0	0	100

Fig. 5. Class population as a percentage of starting population at various times of day.

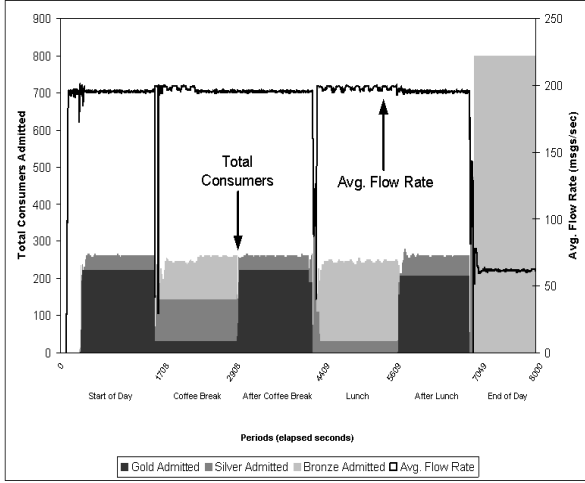


Fig. 6. Total population (stacked) and average flow rate for each stage of the scenario.

We assume that the system only tracks a small number of sectors, i.e., a small number of flows, but the overall event rate may be quite high. In our experiment, we define four flows with identical rates. The minimum rate per flow was 10 msg/sec, and the maximum was 200 msg/sec.

We define three classes of service for each flow: gold class consumers have a utility function of the form $100 \times r$ where r is the allocated rate; silver class consumers have a utility of $10 \times r$; and, bronze consumers have a utility of r . There were a total of 200 gold consumers, 400 silver consumers, and 800 bronze consumers. The consumers were spread equally among the flows.

The trading day is broken up into several segments to simulate variations in population. Table 5 summarizes client load at various times of day. During each segment of the day, we alter the available percentage of consumers of a particular class so that the optimizer is forced to reallocate populations in order to maximize utility. Figure 6 summarizes the results.

As expected, the optimizer favors the available population with the highest utility. For example, when gold consumers are available they are always admitted. Conversely, bronze consumers are only admitted after all available higher utility consumers have been admitted. The differences in utility among classes is sufficiently high that there is no advantage to lowering rate in order to admit more (lower class) consumers.

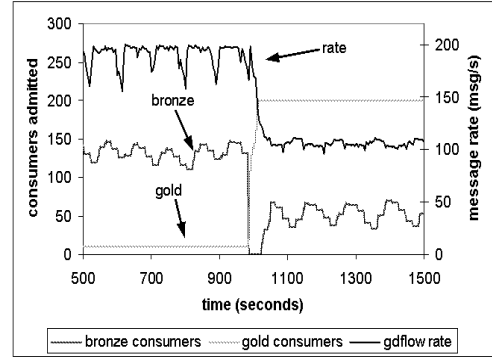


Fig. 7. Population and rate effects when increasing high-priority, high-cost population.

For this reason, message rate remains near r_{max} until the end of the run where only bronze consumers are available and it becomes advantageous to lower rate (i.e., there are no lower class consumers). The periodic dips in message rate occur at segment boundaries because server capacity is temporarily degraded while we manage large changes in population.

In the second experiment, we used two reliable flows and one best-effort flow with the same rate ranges as in the previous experiment. The reliable flows use more resources since they must persist messages to disk. One of the reliable flows (gdfloor) hosts a gold class as in the previous experiment. Similarly, the best-effort flow hosts a bronze class. The remaining reliable flow hosts the same load as the best-effort flow, and is used to help load the system. Once rates stabilized, we increased the number of consumers on “gdfloor” and observed the effects. Figure 7 summarizes the results.

Initially, there are 10 gold consumers on “gdfloor” and approximately 140 bronze consumers on the best-effort flow. Bronze consumers on “gdfloor” are never admitted throughout the test because they have the same utility as bronze consumers on the best-effort flow, but are more expensive to admit since they consume more resources. At time 1000, the available gold population is increased to 200. Since gold consumers have significantly higher utility, the bronze population on the best-effort flow is decreased. However, as reliable flow routing becomes more expensive, e.g., because of higher fanout, internal routing latency increases because messages must queue waiting for access to disk. The higher latency constrains the maximum rate for gdfloor and eventually lowers the rate allocation. As gdfloor rate decreases, there is sufficient capacity to again admit bronze consumers.

Note that bronze consumer population is subject to oscillation throughout the experiment. This is due to the relatively low cost of bronze consumers as compared to available capacity. We are working to improve optimizer output by smoothing changes to population.

B. Excess Resource Capacity

In certain applications, producers may be willing to pay to over-provision the system by specifying a maximum rate

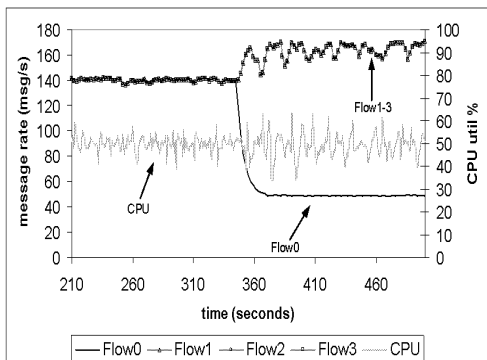


Fig. 8. Rate reallocation due to excess capacity.

that is not always achieved. In this case, there will be spare capacity when the producer’s rate is below the rate allocated by the optimizer. When spare capacity is available, the utility aware rate control component increases utility by adjusting rates on other flows until available capacity has been utilized.

To observe adaptation to excess capacity, we constructed a rate control experiment using four one-way flows each with one consumer class and identical utility. At the beginning of the experiment, all flows produce at their maximum rate (200 msgs/sec) but are allocated a lower rate (140 msgs/sec) due to resource limitations. We then arbitrarily decrease the rate of flow 0 to 50 msgs/sec approximately six minutes into the test. This provides excess capacity which rate control allocates to the other flows. Figure 8 illustrates the rate reallocation.

When the rate of flow 0 is reduced, the rate control component observes a dip in CPU utilization and determines that excess capacity is available. This excess capacity is allocated equally among the flows because each has identical utility. However, only flows 1 through 3 are capable of using the capacity and their rates subsequently rise. The increased rates oscillate because of the “rate snapping” effect described in Section IV. This effect will continue until either flow 0 resumes its previous rate, or its maximum rate target is lowered.

C. Latency Effects

The rate of request-response flows is constrained both by the cost of routing messages for the flows, as well as the latency incurred in processing the flow messages at internal request responders. Thus, the system is forced to make trade offs between admitting consumers on other flows, and increasing the latency of request-response flows. To observe this effect, we constructed an experiment consisting of one best-effort flow with a “gold” subscription class, and one request-response flow (RR flow) with 20 requesting clients. The utility for the request-response flow is identical to the example in Section II, with a think time of 100ms, an execution time of 8ms, a latency threshold of $D = 500$ ms, and a maximum utility of $C = 6000$. This yields a rate range of $r_{min} = 33.33$ and $r_{max} = 203.70$. Initially, there are 200 best-effort consumers

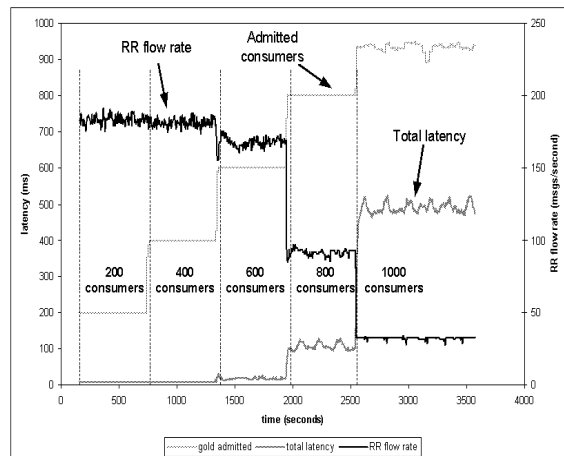


Fig. 9. Trade offs between consumer admission and latency on a request-response flow.

Test	Error Ratio		
	Low	High	Average
Trading Day (Figure 6)	0.67	1.34	1.08
Increase High Utility (Figure 7)	0.34	1.20	1.01
Excess Capacity (Figure 8)	0.99	1.30	1.04
Latency Effects (Figure 9)	0.91	1.09	1.04

Fig. 10. Proportional error correction ranges and averages for each test.

with an additional 200 consumers added every 10 minutes to load the system. Figure 9 summarizes the results.

As shown in the figure, the first 400 consumers have little effect on latency and the rate on the request-response flow remains near 200 msgs/sec. At 600 consumers, latency is marginally increased and request-response rate is slightly reduced. At 800 consumers, latency is significantly increased and request-response rate is halved. Finally, at 1000 consumers the latency threshold takes effect so that at most approximately 900 consumers are admitted and the request-response rate is at r_{min} . At this point, no further consumers will be admitted unless the latency threshold is raised.

D. Model Correction

As discussed in Section III-A, the fit of our CPU models is imprecise and the error (constant term) varies according to flow reliability. Our proportional error correction mechanism compensates for this error using a feedback loop to adjust model constants according to differences in measured and predicted performance.

Figure 10 reports the proportional ranges in our various tests. The error correction ratio is used to scale the CPU utilization targets for the optimizer and rate controller. Despite the error terms, the table shows that the average correction is less than 10% of what our models predict. This increases our confidence in the offline approach we used to develop our models. Tests which were subject to large changes in population (e.g., the first two tests) will tend to skew the range

of corrections since the models do not account for load due to connecting or disconnecting consumers.

VI. RELATED WORK

Resource allocation, both short term, e.g., load balancing, and long term, e.g., provisioning, has been researched in a wide variety of related areas. In this section we discuss related work on short term resource allocation in stream processing systems, systems supporting web workloads, and wide-area communication systems.

A. Stream Processing Systems

Stream processing systems like the Stanford Stream Data Manager [2] and Borealis [3] have their origin in stream databases and focus on providing a declarative approach, like SQL, to specifying the transformations in the system. Resource allocation in this context has looked at using the semantics of relational operators to either selectively degrade the quality of the output, such as by dropping tuples, or parallelize processing on clusters of machines. In contrast, our approach is applicable to an environment where the semantics of transformation are unknown and load is controlled at the boundary of the system by rate and consumer admission control.

Research into stream processing overlay networks has looked into placement of operators in a network of machines [8]–[10], [14], [15]. This work is complementary to ours, which focuses on resource allocation decisions within a node or link, given knowledge of the path of the flow. In [12], we describe a distributed optimization algorithm for making rate and admission control decisions. That work focuses purely on the optimizer component of our architecture (figure 4).

Admission control on a per-message basis is considered in [16], taking into account the reward and penalty for each message, and a centralized solution is proposed. In contrast, we consider aggregate profit metrics, not per message, and perform consumer admission control instead of message admission control.

None of work described above considers request-response flows and how the rate and latency requirements of one-way and request-response flows could be integrated into the same system.

B. Communication Systems

In communication systems, network bandwidth allocation is a problem that has been studied in detail. There has been significant recent work on flow control as an optimization problem, for example [6], [7], [17]. Except for [7], these works have assumed strictly concave utility functions, which excludes piece-wise linear shapes. We extend the algorithm in [7], to handle multiple types of resources, and use it for rate control.

There is also work on multirate flow control algorithms [18], [19] that modify the flow rate, based on knowledge of flow semantics (like video streams), within the system. This work is mainly used for one-to-many (multicast) flows. Only

bandwidth has been considered as a resource, and not the processing cost of changing the rate. We defer the study of multi-rate control in our system context to future work.

Measurement-based admission control of flows has been researched in the context of network resource allocation [20]–[23] to provide QoS guarantees. In contrast, in our work we admit consumers for one-to-many flows assuming all flows are admitted, and exercise rate control on flows. If there is no feasible solution to the optimization problem, i.e., resource bounds are exceeded even at the minimum allocation levels to flows and consumers, flow admission control may be needed and these techniques could be used.

C. Systems for Web Workloads

There is much work on resource virtualization and service differentiation for request-response workloads. In [24], [25], the focus is on service guarantees in terms of desired throughput and latency, rather than defining a continuum of levels using utility functions. In [26], latency guarantees are satisfied using an earliest deadline first (EDF) scheduler, and in [27], a fair queuing scheduler is used to proportionally share resources for throughput, and latency guarantees are secondary. All these approaches use scheduling techniques to provide the guarantees, which rely on two characteristics of such systems that are not true in our context (1) the scheduler is the single entry point to the system, and (2) the scheduler can track when the response is generated for a request, so it has precise knowledge of what requests are outstanding.

In [28], the authors consider utility functions in terms of latency, for different classes of requesters. This is very similar to how the utility functions for request-response flows are initially expressed in our system. Unlike our system, they do not transform the utility functions in terms of latency, into functions in terms of rate. Like other work on request-response workloads, they use a scheduling approach where the scheduler can count the number of outstanding requests in the system.

VII. CONCLUSION

We have described a utility-aware approach for optimizing resource allocation in an event processing system. A key aspect of our approach is the unification of utility models for different interaction patterns. This simplifies optimization and is more complete since rate, latency and consumer population are considered when computing utility. We combine the utility model with experimental resource models and an efficient greedy optimization algorithm to derive utility-maximizing resource allocations.

We have conducted experiments to verify our approach in the context of a single server system. Our experiments show that, in addition to optimizing resources under steady state, our approach can make effective trade offs in response to dynamic system changes. In each case, resources are shifted to favor configurations with optimal utility. We also demonstrated the use of continuous rate control for using slack capacity when available.

Extending our approach to a distributed setting presents many significant challenges. For example, the optimization problem is substantially more difficult to solve with a distributed algorithm. We have developed a distributed algorithm for optimization [12] which demonstrates good scalability but with restrictions on the shapes of the utility functions (strictly-concave, as compared to concave functions assumed in this paper). We are currently developing run-time extensions to support distributed execution.

ACKNOWLEDGMENT

The authors would like to thank the reviewers for their valuable comments.

REFERENCES

- [1] "Java (tm) message service. <http://java.sun.com/products/jms/>."
- [2] "Stanford stream data manager. <http://www-db.stanford.edu/stream/>."
- [3] "Borealis: Second generation stream processing engine. <http://nms.lcs.mit.edu/projects/borealis/>."
- [4] D. A. Chappell, *Enterprise Service Bus*. O'Reilly, 2004.
- [5] S. Shenker, "Fundamental design issues of the future Internet," *IEEE Journal on Selected Areas in Communications*, 1995.
- [6] S. H. Low and D. E. Lapsley, "Optimization flow control I: Basic algorithm and convergence," *IEEE/ACM Transactions on Networking*, vol. 7, no. 6, pp. 861–874, 1999.
- [7] K. Kar, S. Sarkar, and L. Tassiulas, "A simple rate control algorithm for maximizing total user utility," in *INFOCOM 2001*, 2001.
- [8] P. Pietzuch, J. Shneidman, M. Welsh, M. Seltzer, and M. Roussopoulos, "Path optimization in stream-based overlay networks," Harvard University, Tech. Rep., 2004.
- [9] V. Kumar, B. F. Cooper, Z. Cai, G. Eisenhauer, and K. Schwan, "Resource-aware distributed stream management using dynamic overlays," in *Proceedings of ICDCS*, 2005.
- [10] V. Kumar, B. F. Cooper, and K. Schwan, "Distributed stream management using utility-driven self-adaptive middleware," in *Proceedings of Int'l Conference of Autonomic Computing*, 2005.
- [11] E. D. Lazowska, J. Zahorjan, G. S. Graham, and K. C. Sevcik, *Quantitative system performance: computer system analysis using queueing network models*. Prentice-Hall, Inc., 1984.
- [12] C. Lumezanu, S. Bhola, and M. Astley, "Utility optimization for event-driven distributed infrastructures," in *Proceedings of the 26th International Conference on Distributed Computing Systems (ICDCS) (to appear)*, 2006.
- [13] "<http://www.research.ibm.com/distributedmessaging/>."
- [14] J. Shneidman, P. Pietzuch, M. Welsh, M. Seltzer, and M. Roussopoulos, "A cost-space approach to distributed query optimization in stream based overlays," in *Proceedings of NetDB*, 2005.
- [15] Y. Xing, S. Zdonik, and J.-H. Hwang, "Dynamic load distribution in the Borealis stream processor," in *Proceedings of ICDE*, 2005.
- [16] A. Verma and S. Ghosal, "On admission control for profit maximization of networked service providers," in *WWW '03: Proceedings of the 12th international conference on World Wide Web*, 2003.
- [17] F. P. Kelly, A. Maulloo, and D. Tan, "Rate control for communication networks: shadow prices, proportional fairness and stability," *Journal of Operations Research Society*, vol. 49, no. 3, pp. 237–252, 1998.
- [18] K. Kar, S. Sarkar, and L. Tassiulas, "Optimization based rate control for multirate multicast sessions," in *Proceedings of IEEE Infocom*, 2001.
- [19] W.-H. Wang, M. Palaniswami, and S. H. Low, "Necessary and sufficient conditions for optimal flow control in multirate multicast networks," *IEE Proceedings-Communications*, 2003.
- [20] J. Hyman, A. A. Lazar, and G. Pacifici, "Joint scheduling and admission control for ATS-based switching nodes," in *ACM SIGCOMM*, 1992.
- [21] —, "A separation principle between scheduling and admission control for broadband switching," *IEEE Journal on Selected Areas in Communication*, vol. 11, no. 4, 1993.
- [22] S. Jamin, P. B. Danzig, S. J. Shenker, and L. Zhang, "A measurement-based admission control algorithm for integrated service packet networks," *IEEE/ACM Transactions on Networking*, vol. 5, no. 1, 1997.
- [23] R. J. Gibbens and F. P. Kelly, "Measurement-based connection admission control," in *15th Int'l Teletraffic Congress*, 1997.
- [24] T. F. Abdelzaher, K. G. Shin, and N. Bhatti, "Performance guarantees for Web server end-systems: A control-theoretical approach," *IEEE Transactions on Parallel and Distributed Systems*, vol. 13, no. 1, pp. 80–96, 2002.
- [25] J. Zhang, A. Riska, A. Sivasubramaniam, Q. Wang, and E. Riedel, "Storage performance virtualization via throughput and latency control," in *Proceedings of MASCOTS*, 2005.
- [26] C. Lumb, A. Merchant, and G. Alvarez, "Facade: Virtual storage devices with performance guarantees," in *Proceedings of the Conference on File and Storage Technology (FAST)*, 2003.
- [27] W. Jin, J. S. Chase, and J. Kaur, "Interposed proportional sharing for a storage service utility," in *Proceedings of ACM SIGMETRICS*, 2004, pp. 37–48.
- [28] R. M. Levy, J. Nagarajarao, G. Pacifici, M. Spreitzer, A. N. Tantawi, and A. Youssef, "Performance management for cluster based web services," in *Integrated Network Management*, 2003, pp. 247–261.