Control Considerations for Scalable Event Processing

Wei Xu[1], Joseph L. Hellerstein[2], Bill Kramer[1], and David Patterson[1]

[1] Computer Science Dept., University of California, Berkeley, CA
{xuw, pattrsn}@cs.berkeley.edu, kramer@lbl.gov
[2] IBM T.J. Watson Research Center, Hawthorne, NY, USA
hellers@us.ibm.com

Abstract. The growth in the scale of systems and networks has created many challenges for their management, especially for event processing. Our premise is that scaling event processing requires parallelism. To this end, we observe that event processing can be divided into intra-event processing such as filtering and inter-event processing such as root cause analysis. Since intra-event processing is easily parallelized, we propose an architecture in which intra-event processing elements (IAPs) are replicated to scale to larger event input rates. We address two challenges in this architecture. First, the IAPs are subject to overloads that require effective flow control, a capability that was not present in the components we used to build IAPs. Second, we need to balance the loads on IAPs to avoid creating resource bottlenecks. These challenges are further complicated by the presence of disturbances such as CPU intensive administrative tasks that reduce event processing rates. We address these challenges using designs based on control theory, a technique for analyzing stability, accuracy, and settling times. We demonstrate the effectiveness of our approaches with testbed experiments that include a disturbance in the form of a CPU intensive application.

1 Introduction

The advent of the Internet, sensor networks, and peer-to-peer networks has greatly increased the scale of distributed systems, making it more difficult to process events to detect and diagnose problems. Scaling event processing requires an architecture that incorporates parallelism. Herein, we address control challenges in providing such parallelism, namely: (a) providing flow control within replicated elements to avoid overload conditions and (b) balancing load in the presence of variable processing demands and other disturbances. Our solution to both of these challenges employs control theory, a formal approach to designing feedback loops.

Event streams consist of many kinds of data. For example, there are notifications of requests for service such as requests to a DNS (Domain Name Service) for name resolution; performance statistics such as response times; and trouble tickets that describe actions taken. These data are input to event processing components that detect abnormal situations, anticipate future problems, and diagnose existing problems.

Our motivation for scaling event processing comes from a company that is key to the eCommerce ecosystem. At the heart of this business is a DNS root server that generates events at a rate of 11 million to 42 million per hour. Many off-the-shelf products provide event processing capabilities, such as HP OpenView[5], IBM Tivoli[6] or Microsoft Operations Manager[7]. However, all are severely challenged by such high event rates.

Much related work exists in the area of event processing. Yemini *et al.* consider how to associate problem causes with symptoms using a code book algorithm [15]. Hofmeyr *et al.* develop techniques that discriminate between normal and abnormal operations [10]. Pinpoint System deals with the localization of failures on a production eCommerce system based on decision trees that analyze event data [3] and further extended to detect anomalies and failed components by automated analysis of execution paths in J2EE(Java 2 Enterprise Edition) applications [4]. Vilalta *et al.* predict critical events in computer system such as high CPU utilization and router failures by applying temporal data mining and time series analysis [14]. Burns *et al.* describe how to construct processing rules from event data [1]. These results identify requirements for event analysis, such as the need to have events in time serial order and to estimate accurately statistics such as the distribution of event sources and response times.

Supporting large scale event processing requires a scalable infrastructure. Astrolabe [13] and PIER [11] provide scaling by collecting and analyzing data on the nodes where they are generated. However, this approach limits the scope of the events analyzed to a single node. The Siena system [2] provides a publish/subscribe event-notification service with considerations for efficiencies and scaling. However, since this is a general infrastructure, it does not exploit the characteristics of event processing such as the opportunity to do intra-event processing in parallel.

From the foregoing, it seems that there has been little focus on scaling event processing. Thus, we introduce an approach that provides scaling through parallelism by identifying two kinds of processing that take place in event processing. Inter-event processing, such as problem diagnosis, analyzes multiple events in combination. Intra-event processing, such as filtering events from specific sources, considers events in isolation. Intra-event processing is easily parallelized by replicating the elements used for intra-event processing. We refer to these as intra-event processing elements (IAPs). We have encountered two challenges in scaling intra-event processing. First, IAPs are subject to overloads that require effective flow control, a capability that is often missing in off-the-shelf components. Second we must balance the load placed on IAPs to avoid bottlenecks. These challenges are further complicated by the presence of disturbances such as CPU intensive administrative tasks (e.g., Java Virtual Machine (JVM) garbage collection) that reduce event processing rates.

The remainder of this paper is organized as follows. Section 2 presents our architecture for scalable event processing. Section 3 applies control theory to designing key elements of a scalable event processing system. Section 4 reports the results of experiments we conducted to assess scaling. Our conclusions are contained in Section 5.

2 Architecture

This section describes our architecture for scalable event processing.

Event processing operates on the attributes of events and the relationships between these attributes. For example, performance events may have attributes such as *IP address*, *memory utilization*, *swap utilization*, and *load average.* For example, an event processing system might employ rules (or other representations) such as the following:

- Rule 1: Discard performance events from the subnet 92.126.10/24.
- Rule 2: Send an alert if the largest load average exceeds 2 for the hosts on subnet 92.126.11/24.

Rule 1 might be used to filter events from a test machine. Rule 2 is useful if all machines on the subnet 92.126.11 are production servers and we want to determine if there is a resource bottleneck.

Rules 1 and 2 suggest that there are two kinds of event processing: (1) intra-event processing such as filtering events that are not of interest and (2) inter-event analysis such as detecting a resource bottleneck. By definition, intra-event analysis is done on events in isolation. Inter-event analysis establishes relationships between events and so typically processes events in time serial order.

In the sequel, we focus on intra-event processing because of the opportunity to scale event processing by distributing the work to multiple nodes. Examples include: sampling events to obtain a representative distribution of response times; cleansing data to eliminate ill-formed events and unnecessary attributes; and augmenting events to associate the host name and host type based on host IP address. Through we only look at one event at a time, the processing can be expensive. We see that many of these operations require access to other information sources, such as a table that relates IP address to host name and type.

These observations led us to the two tier architecture that is depicted in Figure 1. Incoming events arrive in (rough) time sequence order. The first tier provides scalable intra-event processing, such as projections to eliminate unwanted attributes and joins to include additional attributes. There are three types of elements in this tier: the load splitter, the **I**ntr**A**-event **P**processing elements (IAP), and the combiner. Scaling is provided by having multiple IAPs that operate in parallel, possibly with different processing speeds and other characteristics. The load splitter assigns events to an IAP, and the combiner consolidates the results in time sequence order. We must be sure that we can implement the load splitter in a very efficient way so that it is not a bottleneck. As we show in Section 3.2, because the load splitter does not look into the event, it is much faster than the IAPs.

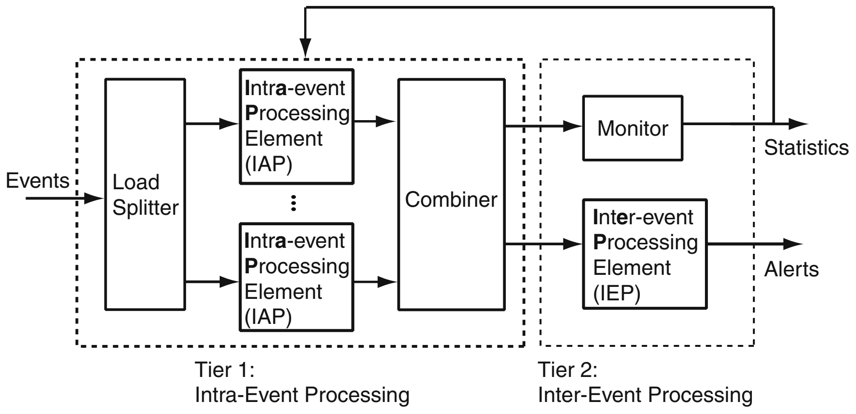
Events
Load Splitter
Intra-event Processing Element (IAP)
Intra-event Processing Element (IAP)
Combiner
Monitor
Statistics
Inter-event Processing Element (IEP)
Alerts
Tier 1: Intra-Event Processing
Tier 2: Inter-Event Processing

Fig. 1. A two tier architecture for scalable event processing. The first tier processes events in isolation. The second tier addresses relationships between events. Scaling is achieved in the first tier by having multiple IAP elements.

The second tier in Figure 1 performs inter-event processing. The **I**nt**E**r-event **P**rocessing element (IEP) inputs events in time serial order, and outputs alerts and higher level events. The monitor calculates statistics that are used as filtering criteria by the first tier, such as quantiles of response times contained in the attributes of incoming events that are used to identify exceptional situations.

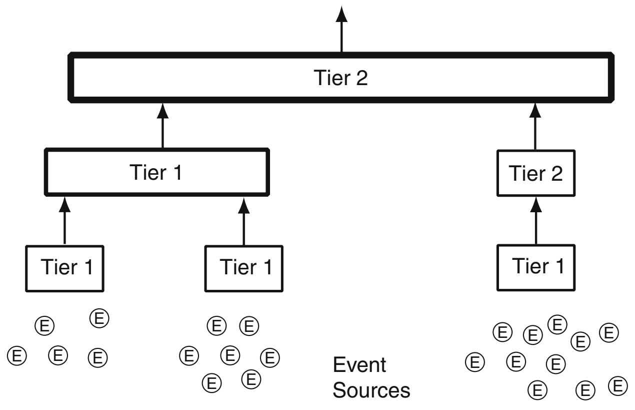

Fig. 2. Generalized architecture for scalable event processing

Figure 2 generalizes the architecture in Figure 1 to handle large scale distributed systems by treating tier 1 and tier 2 as components that can be replicated as needed. For example, network utilizations and communication delays are reduced by filtering events close to their origin. This argues for having instances of the first tier in many locations, such as satellite campuses and local area networks for critical servers. In contrast, a second tier instance may be quite distant from event sources in order to have a sufficient scope of events to do root cause analysis and obtain accurate statistical distributions. Thus, multiple first tier instances may feed into a single second tier. It may also be that there is a hierarchy of second tier instances, such as for event processing that occurs based on geographic scale (e.g., city, state, country). Thus, multiple second tier instances may input events to another second tier instance.

We focus on the requirements for scaling in the first tier. To better understand these issues, we implemented an IAP that embeds a TelegraphCQ (TCQ) system [12] to handle SQL based processing of event streams within the IAP. Our studies reveal two issues with increasing the event input rate. The first issue relates to flow control within IAP nodes. The second concerns balancing the loads of the IAPs.

3 Control Design

This section describes how we address issues in scaling intra-event processing, namely—(1) flow control within IAP nodes and (2) load balancing across IAPs.

3.1 Flow Control Within IAP Nodes

We begin by studying the effect of load on an IAP. Since our IAPs embed a TCQ, we represents events as data tuples in a object-relational schema and quantify throughput by using the TCQ metric tuples/sec, which is the same as events/sec. Figure 3 reports the results of experiments conducted on a single IAP node. We see that at moderate to heavy loads, tuples are dropped. This is problematic for two reasons. First, drops are not selected at random and so the presence of drops can bias the event statistics that are used for threshold-based filters and other purposes. Second, as we can see in the next paragraph, the drop happens after all processing on that tuple is done. Thus, dropping a tuple does not reduce workload on a server.

Going into more detail, a TCQ is structured into two parts: (a) a front-end process that interacts with requesters, parses inputs and translates them into internal data structures and (b) a set of back-end processes that perform relational database operations. The output of the back-end is placed into a *result queue* whose entries are retrieved by the front-end to respond to in-coming requests. The drops are a consequence of an overflow of the result queue, which is evident in Figure 3 since drops occur as the free space goes to 0.

One solution to the drops problem is to have front-end processes block when the result queue is full. Thus, the tuple is either dropped or throttled via admission control. Unfortunately, this can cause unpredictable effects on other queries because of the complex sharing that takes place. A second approach is to make the result queue very large to avoid having drops. But this means there is less memory available for front-end and back-end processes, which reduces throughput. A third technique is to do off-line experiments to determine the processing

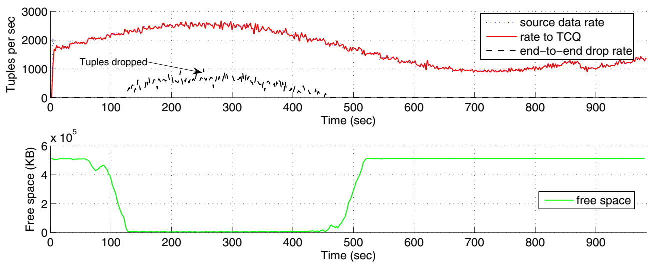

Fig. 3. Behavior of a TCQ node without regulating result queue length. The top plot shows the event input rate, and the drop rate. The bottom plot depicts the free space in the result queue. The drop rate increases with the event input rate.

capacity of an IAP node for a representative set of events. However, this is difficult to do because certain dynamics affect free space of the result queue, such as changes in the distribution of event types that in turn affects the amount of processing done (especially due to the selectivity of database queries).

Our approach to eliminating drops is to implement flow control within the IAPs by regulating the rate at which events are accepted by front-end processes. Thus, events are held in a queue within the IAP until there is sufficient space in the result queue. Such a design avoids the complexities of blocking front-end processes that hold resources associated with partially completed requests.

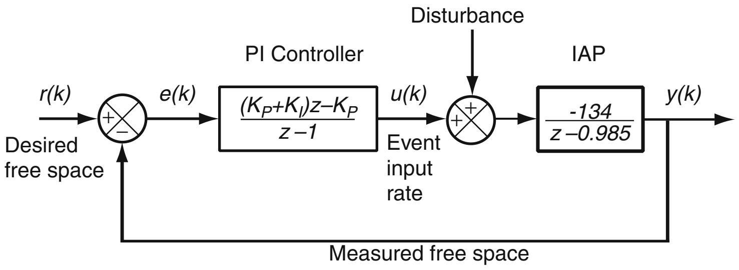

Fig. 4. Block diagram of IAP admission control

Figure 4 is a block diagram of the IAP flow control that we propose. This control system seeks to maximize throughput without dropping events by regulating the free space of the result queue. The reference input is the desired free space. The controller uses the difference between this reference and the measured free space of the result queue to adjust the event input rate.

We design the controller as described in [9]. The first step is to model how the event input rate affects free space of the result queue. Let $y(k)$ be the free space at time k, and let $u(k)$ be the event input rate. We use the first order model

$$y(k+1) = ay(k) + bu(k) \tag{1}$$

since [9] contains many examples in which this works well for real systems. Values of the parameters a and b are estimated from data obtained from studies of a testbed system, yielding $a = 0.985$ and $b = -134$. With this, we express the relationship in Equation (1) as a *transfer function*, a representation that expresses time serial effects in terms of z (which can be interpreted as a delay operator). The transfer function here is

$$\frac{-134}{z - 0.985}. \tag{2}$$

The transfer function in Equation (2) provides several insights. First, consider the transfer function's *steady state gain*, a quantity that indicates the effect of a small change in event input rate on free space. Steady state gain is obtained by evaluating Equation (2) at $z = 1$, which is -3,190. Having a negative steady state gain means that free space declines as the event input rate increases, which is consistent with intuition. A second insight from Equation (2) relates to its *poles*, the values of z for which the denominator is zero. Equation (2) has a single pole at 0.985. The poles of the transfer function must lie within the unit circle of the complex plane for the system to be stable, which is the case for this transfer function. Further, poles that are closer to the unit circle indicate a system with a longer settling time (convergence time). From [9], settling time k_s is approximately

$$k_s \approx -4/ln|a|. \tag{3}$$

Applying Equation (3) to Equation (2), we determine that the open loop settling time is approximately 264 sec. That is, if there is a transitory change in the event input rate, it will take the IAP 264 sec to return to its previous state.

We design the controller with two objectives in mind. First, we want to accurately regulate free space. Second, we want to minimize the effect of *disturbances* such as changes in the types of events and the execution of administrative tasks (e.g., garbage collection) on IAP nodes. We employ proportional-integral (PI) control, an approach that is widely used because it ensures that the measured output converges to the reference input, and a PI controller is easy to understand and implement. The control law for a PI controller is:

$$u(k) = u(k-1) + (K_P + K_I)e(k) - K_P e(k-1) \tag{4}$$

where K_P and K_I are controller parameters that are determined by design.

We want the controller to settle (converge) quickly and so choose as our objective that the closed loop settling time should be 5 time units. This is achieved by properly choosing the parameters K_p and K_I. The first step is to invert Equation (3), yielding $a \approx e^{-k_s/4}$. For $k_s = 5$, $a \approx 0.449$. Next, we derive the denominator of the transfer function of the closed loop system, which is the polynomial $z^2 - (134K_P + 134K_I + 1.985)z + 134K_P + 0.985$. Setting the poles of the polynomial according to k_s and a, we get $K_P = -0.00614$ and $K_I = 0.02168$ causes this polynomial to have zeros at $0.046 \pm 0.4i$. Since $|0.046 \pm 0.4i| \approx 0.4$, the closed loop system has a settling time of approximately 5 time units.

3.2 Load Balancing Between IAPs

Load balancing provides a way to reduce response times by reducing the utilization of bottleneck resources, those resources that largely determine the response time of a system. Load balancing is particularly important in parallel computation systems involving synchronization because having an imbalance in processing speeds causes faster nodes to wait for slower nodes. In our system, the combiner is a barrier coordinator. A slow IAP forces the combiner to wait, slowing the progress of events to the second tier. The load balancer must be efficient enough to avoid being a bottleneck. It must decide where the event should go upon event arrival. It does not have time to route based on the content of the event, or hold events in a buffer for delayed decision making.

Let $\mathbf{L} = (L_1, \cdots, L_N)$ be the load in events/sec applied to the N IAPs, and let $R_1, \cdots, R_N$ be their response times at these loads. One way to formulate the objective of load balancing is to find $\mathbf{L}$ that minimizes $\sum_i (R_i - \bar{R})^2$ (where $\bar{R}$ is the average response time) subject to the constraint that $\sum_i L_i$ is constant. Unfortunately, this is a non-linear optimization that is quite complicated to solve.

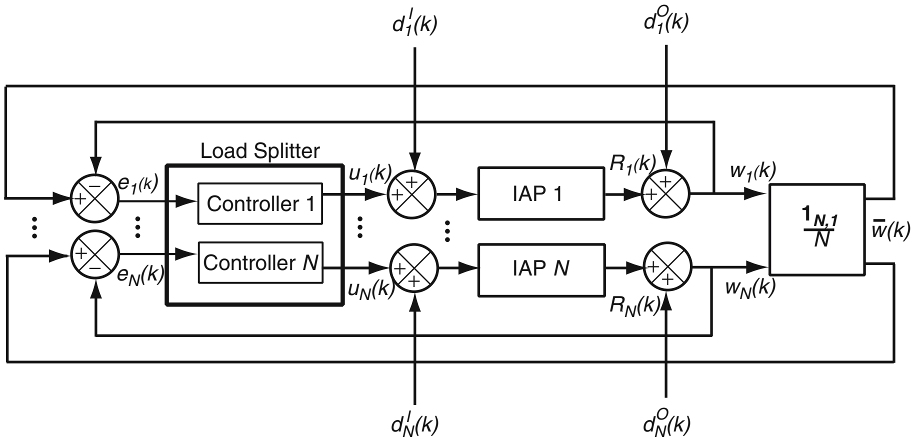

Fig. 5. Block diagram of a load balancing controller

Figure 5 depicts an approach to load balancing based on classical control theory. Employing the results in [8], load balancing is accomplished through regulatory control by having the reference input be the mean value of the (disturbance adjusted) response times of the IAPs. Such an approach allows us to regulate the IAPs so that they converge to the same value, the mean response time. In the studies we conduct in Section 4, only two IAPs are used and so the foregoing is simplified in that we need only to regulate the difference in the outputs from the two IAPs (a design that requires only one controller in the load splitter).

For small values of N, the controller can be derived in a manner similar to that done in the last section. However, for larger N, more sophistication is required. A control theory technique that is well suited to this situation is linear quadratic regulation (LQR). LQR provides a framework for constructing optimal controllers. [8] describes how to use LQR for the block diagram in Figure 5.

4 Experiments

This section describes experiments conducted to assess the control designs in Section 3.

First, we evaluate the effectiveness of the flow control system in Section 3.1 for regulating free space of the result queue. The controller is implemented as the manager of an input buffer. Every 2 sec, the controller reads the TCQ log to obtain the current size of the result queue, and uses the PI control law to calculate the number of events to send to the IAP over the next 2 seconds. Tuples that are not sent remain in the input buffer.

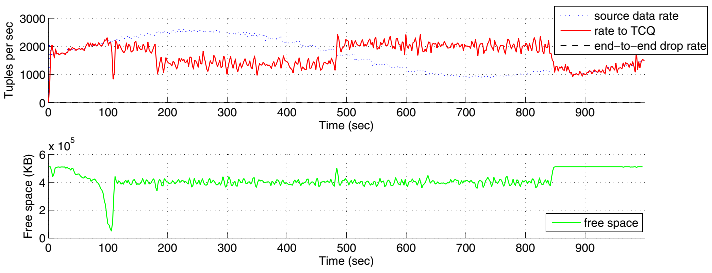

Fig. 6. Regulation of TCQ free space using a PI controller. The reference input is 400 MB. A CPU hog is introduced at time 180. The system quickly adapts the input rate so that free space returns to 400 MB.

Figure 6 plots the results of an experiment in which the reference input for the result queue is 400 MB. In the figure, the decline in free space at 100 sec is the result of a TCQ start-up effect that the controller corrects in a very short time. An input disturbance in the form of a CPU intensive application (hereafter CPU Hog) is introduced at time 160 sec, which causes a reduction in the TCQ throughput. However, when the CPU Hog completes at time 500, TCQ throughput returns to its previous level. Note that from time 100 sec through 500 sec, the input load on the TCQ is greater than its maximum capacity. Even so, the controller maintains the free space at 400 MB, and the excess load is held on the input queue for further processing, such as load balancing.

Next we assess the load balancing controller described in Section 3.2 for use as the load splitter in Figure 1. The testbed developed for this assessment is

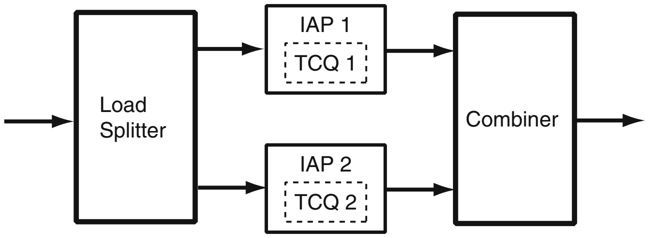

Fig. 7. Testbed used to evaluate the load balancing controller used as a load splitter. Boxes with solid lines are separate dual CPU computers with 1.5GB of RAM.

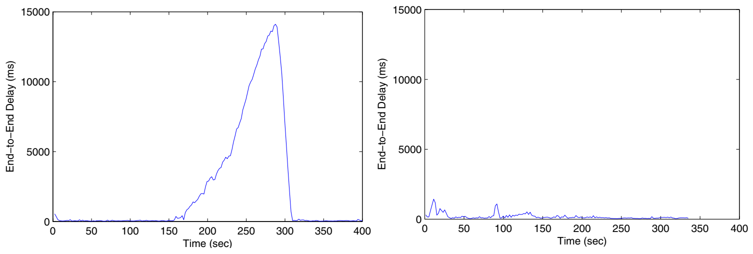

Fig. 8. LEFT: Average delay using a round-robin scheme to assign events to IAPs. RIGHT: Average delay using a well designed controller.

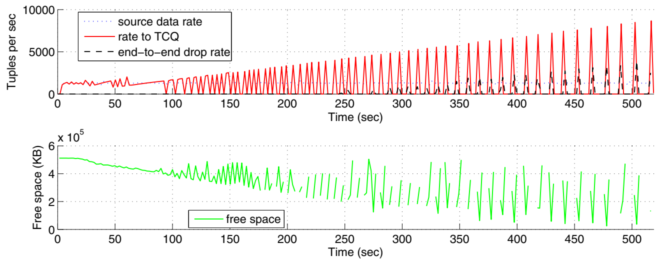

Fig. 9. Performance of an incorrectly constructed load balancing controller

depicted in Figure 7. There are two IAP nodes, and all boxes with solid lines indicate separate dual CPU computers with 1.5GB of RAM.

Figure 8 (LEFT) plots response times using a round robin scheme for the load balancing controller. This assessment includes a disturbance in the form of a CPU Hog that is started on one IAP node at time 160. We see that end-to-end delays grow to be quite large. In contrast, Figure 8 (RIGHT) plots response times when the load balancing controller is used. As in the plot on the LEFT, a CPU Hog is started on one IAP node at time 180. The load balancing controller handles this disturbance well, moderating the impact of this disturbance so effectively that there is almost no detectable change in end-to-end delays.

Last, we describe an experiment that we conducted by accident. Figure 9 plots the end-to-end delays for the system in Figure 7 using an incorrectly designed flow controller. We see that the system is *unstable* in that there are oscillations that increase in amplitude with time. At first, these characteristics seemed to be inconsistent with our control analysis, which predicted a stable system. Since control theory derives the controller from the model of the target system, we re-visited the model and discovered that it failed to consider that control actions take place in the next time interval. Correcting our models and re-designing the flow controller resulted in the performance displayed in Figure 6.

5 Conclusions

Scaling event processing requires an event processing architecture that incorporates parallelism. Intra-event processing is easily parallelized. We propose an architecture to support this parallelism in which intra-event processing elements (IAPs) are replicated to scale the system to larger event input rates. We address two challenges in this architecture. First, the IAPs are subject to overloads that require effective flow control. Second, we need to balance the load placed on processing elements to avoid resource bottlenecks. These challenges are further complicated by the presence of disturbances such as administrative tasks (e.g., garbage collection) that reduce event processing rates. We employ control theory to address both challenges since control theory provides a systematic approach to design that includes considerations of disturbances. Our solution for the flow control problem is based on regulatory control of the free space of the result queue, a key resource in our IAP implementation. Our solution for load balancing employs a technique that transforms an apparent optimization problem into a regulatory control problem. Studies done on a testbed system show that our control designs provide good performance under time varying loads and disturbances in the form of a CPU intensive application.

One area of future research is to explore the use of techniques from adaptive control and statistical learning theory to deal with stochastics and nonlinearities that are more difficult to address with classical control theory. Another direction is to expand the set of scaling experiments to gain more insight into the limitations of our current designs. Last, we want to make our IAP system available to system operators and researchers so that others can benefit from our work.

Acknowledgements

This work is in part supported by NSF grant number CNS-0509559 and Micro award number is 04-072.

References

1. L Burns, JL Hellerstein, S Ma, CS Perng, DA Rabenhorst, and D Taylor. A systematic approach to discovering correlation rules for event management. In *IEEE/IFIP Integrated Network Management*, May 2001.

2. Antonio Carzaniga, David S. Rosenblum, and Alexander L. Wolf. Achieving scalability and expressiveness in an internet-scale event notification service. In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 219–227, Portland, Oregon, July 2000.
3. Mike Chen, Alice Zheng, Jim Lloyd, Michael Jordan, and Eric Brewer. A statistical learning approach to failure diagnosis. In *International Conference on Autonomic Computing (ICAC-04), New York, NY*, May 2004.
4. Mike Y. Chen, Emre Kiciman, Eugene Fratkin, Armando Fox, and Eric A. Brewer. Pinpoint: Problem determination in large, dynamic internet services. In *DSN*, pages 595–604, 2002.
5. Hewlett-Packard Development Company. Hp OpenView. http://www.openview.-hp.com/, 2005.
6. IBM Corporation. Tivoli. http://www.ibm.com/software/tivoli/.
7. Microsoft Corporation. Microsoft Operations Manager. http://www.microsoft.com/mom/.
8. Yixin Diao, Joseph L. Hellerstein, Adam Storm, Maheswaran Surendra, Sam Lightstone, Sujay Parekh, and Christian Garcia-Arellano. Using MIMO Linear Control for Load Balancing in Computing Systems. In *American Control Conference*, pages 2045–2050, June 2004.
9. Joseph L. Hellerstein, Yixin Diao, Sujay Parekh, and Dawn M. Tilbury. *Feedback Control of Computing Systems*. Wiley-IEEE Press, Aug 2004.
10. Steven A. Hofmeyr, Stephanie Forrest, and Anil Somayaji. Intrusion detection using sequences of system calls. *Journal of Computer Security*, 6(3):151–180, 1998.
11. Ryan Huebsch, Joseph M. Hellerstein, Nick Lanham, Boon Thau Loo, Scott Shenker, and Ion Stoica. Querying the internet with PIER. In *Proceedings of the 29th VLDB Conference*, 2003.
12. Sailesh Krishnamurthy, Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Samuel Madden, Frederick Reiss, and Mehul A. Shah. Telegraphcq: An architectural status report. *IEEE Data Eng. Bull.*, 26(1):11–18, 2003.
13. Robbert van Renesse, Kenneth P. Birman, and Werner Vogels. Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. *ACM Transactions on Computer Systems*, 21(2):164–206, 2003.
14. Ricardo Vilalta, Chidanand Apté, Joseph L. Hellerstein, Sheng Ma, and Sholom M. Weiss. Predictive algorithms in the management of computer systems. *IBM Systems Journal*, 41(3):461–474, 2002.
15. S. A. Yemini, S. Kliger, E. Mozes, Y. Yemini, and D. Ohsie. High speed and robust event correlation. *IEEE Communications Magazine*, 34(5):82–90, 1996.