ABSTRACT

Processing log data from a large distributed system is challenging. The collection and analyzing such data requires timely and accurate pre-processing and sanitizing the data, which is best done with a stream processing system. This paper explores ways to add control theory to a streaming log processing system to improve its accuracy and throughput in the face of varying loads and disturbances. We introduce a way to regulate the internal queue occupancy, preventing data loss. We also provided an accurate and reliable data source to feed the analysis system using feedback control. We compare the results of two different types of commonly used controllers, Proportional Control (P Control) and Proportional-Integral Control (PI Control). We discussed our experience in different stages of applying control theory to a practical system and showed the benefits of designing a system on a solid theoretical foundation.

1. INTRODUCTION

1.1 Motivation

An increasing number of large complexes of systems are being used today to support a range of activities from processing web transaction processing data to computationally intensive simulation. All such complexes generate a large number system logs that need to be processed for a number of reasons, especially for automatic detection of failures in the system[10].

One practical problem is that system logs are huge and generated very rapidly. For example, the data set we used in this project, which is request log from a Domain Name Service (DNS) root server has 11 million to 42 million records per hour. Data logs a large not only because of the number of records, but because each record can have hundreds of attributes. Because logs are continuously generated, the value of older logs are decreasing quickly if not processed in a timely manner[4].

To address problems such as above, one of us [11] designed a software architecture that processes system logs as data streams. The system make use of TelegraphCQ[9], a general purpose system for processing data streams, as building block, and allows parallel and multi-tier processing of stream data discussed below. It provides a simple way to specify the system infrastructure and running pre-processing on a cluster.

However in the processing of testing and using the system, we found a major problem with the system is that we one of the n node running parallel processing of tuples got slow, which happens often on a cluster due to various disturbances, the whole system becomes slow, waiting for the slowest node. This is very undesirable since modern computer clusters can get different disturbances, both software and hardware that cause the performance to degrade for a short period of time[2]. Also, current implementation of TCQ drop tuples silently if result are not pulled from its output buffer fast enough, as described in section 3.

Control theory provided powerful tools for analyzing and regulating system dynamics. It has the advantage over ad-hoc design of addressing software system dynamic issues in the sense that it produces more predictable result in under different disturbance conditions of the system. It has been applied to many scenarios in computer systems.

The challenges of applying control theory to computer systems involves dealing with non-liberality and stochastics of computer systems[5]. Also, a practical challenge is that it is hard to reduce a computer problem to control problem which involves discovering the parameters that can be controlled (the nobs that can be turned) and the quantities that can be monitored and used in the controller. This problem is especially serious in systems that is not designed to be measurable and controllable. As an important part of our future work, we want to come up with some guideline that helps designing systems that is easy to monitor and control.

In this project, we used a somewhat different point of view, we took the legacy system that is not designed controllable for robustness, and by applying control theory, we make the behavior of the system more desirable. Throughout the process, we treat the target system as a black box and all changes are implemented with only a few sensor instrumentation to the target system.

Later parts of the paper is organized as follows: In section 2 we briefly describe the background information and related work. In section 3, we define the control problem in detail, and we review the process of estimating the parameters of the target system and the controller design in section 4. Experimental results are described in section 5 and in section 6, we discuss some practical experience on applying control theory to computer system, and show how powerful when this theory is applied to computer systems. We conclude and discuss about our future work in section 7 and section 8, respectively.

2. BACKGROUND AND RELATED WORK

In this section, we make a brief introduction to TCQ, the target system we are controlling. We also discuss related
work in control theory and cluster load balancing.

2.1 Introduction to TCQ

General stream processing techniques have been studied in database community in great depth [1, 4]. A number of general purpose stream processing systems have been built [3, 8]. We use Telegraph Continuous Query engine [8], which is designed to process data streams with adaptive, continuous queries.

The queries are specified in PostgreSQL SQL, with all data types and functions. One query is usually specified by a few lines of SQL and all the query plans are automatically optimized. This makes adding and modifying queries significantly easier than ad-hoc scripts.

We believe building a system based on TCQ can make implementing both statistical learning theory (SLT) algorithms and general purpose system monitoring and data collection easier. However, in order to handle large amount of data, we need to run TCQ both in parallel and in multiple tiers.

The developers of TelegraphCQ are building a similar multi-tier system, which allows querying on a large number of streams coming from geographically distributed and independently managed data sources [9]. This multi-tier system is especially useful in collecting data from sensor networks [7].

2.2 Control Theory in Computer Systems

Control theory has a long history going back over 2000 years to early Greek and Arab engineers. [5] provides an interesting history of the evolution of control theory evolving through the industrial revolution to modern machines. Applying control theory to computer systems is also not new, but is increasing in importance. In computer systems, the output of a systems depends on the type and number of requests or actions a system must perform, which is called the workload. Workload is characterized as the arrival distribution of service requests, and the distributions of the service times for the resources. [6] provides an extensive review of the current techniques for applying control theory to computer systems.

2.3 Other related work

[2] addressed a similar problem of node slowing down in parallel database systems. Their solution is based on a protocol so the data sender and receiver can agree on the data rate to be sent. In this way, the system never overhears, but the implementation is complex and requires changing on both the data sender and the receiver. While a system using feedback control allows control error to happen and try to correct it afterwards. This allows us to deal with complex target system as a blackbox.

Applying control theory to computing system requires collecting data describing system performance efficiently. A lot of data are currently collected, and several scalable network monitoring tools can be used for this purpose, even in large computer clusters. Example of these monitoring tools includes: HP OpenView (http://www.openview.hp.com/), IBM Tivoli (http://www-306.ibm.com/software/tivoli/), Microsoft Operations Manager (http://www.microsoft.com/mom/).

Our system can be used by data mining community for other data mining problems as well [4].

3. CONTROL PROBLEMS

In this section we describe in detail the problem we try to address in the stream processing system using control theory.

3.1 Providing an accurate data source

It is important that we can accurately control the work load of target system. In this experiment, we used a data source, which read input data from a file on disk, put the data into its own buffer, and send out the data at a desired data rate. The desired data rate is specified as a function over time.

However, without explicit control, due to thread scheduling and network latency, the actual data output rate is always smaller than the desired data rate (see Figure 1). If we only care about the relative performance with different workloads, this difference does not matters much. However, in our case, the operation range is important since we have to make sure that the system works in its targeted range. We also believe that this problem is general for many other systems emulations.

Since the oscillation comes from uncontrolled disturbance in computer systems, such as scheduling and competing for resources such as disk or network bandwidth, it is hard to find a feed-forward solution for these systems. However, feedback control provides a simple solution.

We take the desired data rate as reference input ($r$) and measure the output data rate (an average over the last 2 sec.) as output ($y$). The 2 seconds interval is a magic number that is consistently used on all data measurements throughout the whole system. It is obtained through experiments and observing the tradeoff between accuracy and smoothness of the observed changes over time.

The controller computes the control errors, and calculates control input $u$ and send it into the output thread as the desired data rate. Note that $u$ is a data rate that is higher than reference input, which "tricks" the output thread into producing a data rate that is lower than $u$ (as it naturally does) but hopefully equals the desired data rate $r$. Thus
the gain of the close loop transfer function is zero (e.g. the output $y$ equals to reference input $r$, meaning that the controller “cancels” the effect of transfer function modelling the output thread).

3.2 Controlling the result queue length on TCQ

While testing TCQ with a larger data rate, we found the following behavior of the system. TCQ system is structured as a multi-tier multi-process system. One data tuple goes through the following sequence of processing in the system. 1) Wrapper cleaning house, which parse the tuple and translate it into the internal data structure of TCQ. 2) TCQ backend, where the tuple is processed by multiple relational operators, including projection, selection, join and aggregation. 3) the result of this processing is sent to a result queue, waiting to be fetched a frontend process. The frontend then pass the tuple to the application[8, 9].

The problem arises when the first and second stage can process tuples faster than the frontend process can pull tuples from TCQ. This is an undesirable property in that it is very hard for the result queue to apply back pressure on the earlier stage of the system, since the backend is complex and the operators are often shared by multiple queries running in the system (blocking one operator may affect the performance of an query that does not use the result queue at all).

To address the problem, we need to regulate the free space on the result queue from getting to zero by adding an input buffer in front of TCQ wrapper cleaning house (stage 1 as described in the last paragraph). Of course the input buffer should be controlled so it will pass the correct data rate into TCQ to make (conservatively) a safe amount of free space on the result queue.

An open loop system requires the understanding of both current workload, including the data rate and the selectivity (i.e. what percentage of tuples will make it to the result queue before being discarded) and the disturbance of the system (e.g. CPU contention). However, both of them are impossible to obtain in advance.

A Feedback control system solves this problem by monitoring the queue length. No matter what reason (selectivity change or CPU contention), caused the queue length not to be at the desired value, the buffer controller will take action to change the data input rate into TCQ to the value that helps to restore the amount of free space.

The reference input $r$ of the controller is desired free space on the result queue, the output $y$ is the queue length, and the controller calculate control input $u$, which is the data rate sending to TCQ node from buffer.

4. PARAMETER ESTIMATION AND CONTROLLER DESIGN

In this section, we briefly describe the methodology of estimating the parameters of the target system and designing of the controller. This process follows closely the book [5].

4.1 Parameter Estimation

Since we treat the target system as black box, the first step is to estimate parameters for constructing transfer functions of target system. To make things simple, we use first order system model, in which the output of target system is defined as:

$$y(k + 1) = au(k) + by(k)$$  \hspace{1cm} (1)

Our task is to estimate the parameters of $a$ and $b$.

We first review the general technique for parameter estimation and then describe the implementation of these techniques in the two target systems described in 3.

We first conducted experiments to collect data for $u(k)$ and the corresponding $y(k + 1)$. This is not easy in practice, especially for systems whose output $y$ is an integral of inputs ($e.g.$ queue length). It is difficult because the analysis involves providing a carefully designed and varying work load in order to make sure the system is working in not saturated range (in which the system behaves linearly).

The raw data are in the format of $N_x2$ matrix, where $N$ is the total number of data points collected. Each line of data points is in the format of $[u \ y]$.

We normalized the data points around their operation point (in which case is the mean of the data set), and get:

$$u = u - mean(u) \ (1 : end - 1)$$
$$y = y - mean(y)$$

In MATLAB, we construct the data set as:

$$X = [u(1 : end - 1) \ y(1 : end - 1)]$$
$$Y = [y(2 : end)]$$

and run a standard procedure in MATLAB that does least square regression to solve for $a$ and $b$ in the following equation:

$$(a \ b) \ast X = Y$$

Plotting the predicted value of $y(k + 1)$ with estimated $a$ and $b$ confirms that they are accurate.

4.1.1 Parameter Estimation for the Data Source

In this case, output $y$ is the actual output data rate of the data source and $u$ is the desired data output. As shown in figure3, the desired and actual data rate are almost linearly related for a large operation range and we choose the operation point at 2140 tuples/sec for input $u$, and the corresponding operation point for output $y$ is 1775.

Following the procedures described above, we get the model for data source:

$$y(k + 1) = -0.59598u(k) + 0.8451y(k)$$

Or expressed in $z$-domain as a transfer function as:

$$\frac{0.8451}{z + 0.5998}$$  \hspace{1cm} (2)

4.1.2 Parameter Estimation for Queue Length Controller

As described in 3, the $y$ value of this system is the observed TCQ result queue length, while the input is the number of tuples sent to TCQ node per second. We implemented system that allows the free space varying within the range of 512MB and 150MB, and ignores all the points where the queue is full (saturated situation). The collected data points and the result of linear regression is shown in figure3.

And the model we get for this system is:

$$y(k + 1) = 0.9845u(k) - 134.31y(k)$$
Figure 2: Behavior of TCQ node with out any control to regulate the result queue length. The top plot is the source data rate, the middle one is the output rate of TCQ input buffer (i.e. number of tuples entering the TCQ processing engine, and the bottom plot shows the free space on the result queue (queue length is the max queue length minus the length of result queue). In this case, though the average data rate is less than the max throughput of TCQ, tuples still get dropped since the result queue fill up. This is because the input buffer does not know it is providing a data rate larger than the subsequent stages of TCQ can handle. Thus a controller is necessary to regulate the buffer output rate.

Or expressed in z-domain as a transfer function as:

\[
\frac{-134.31}{z - 0.9845}
\]

(3)

4.2 Controller Design

We explored the use of both Proportional Controllers (P Controller) and Proportional-Integral-Controllers (PI Controller). Both are standard controller designs in control theory.

The advantage of P Controller is that it is relatively simple to design, since its control input is simply control error multiplied by a constant \( K_p \). The control law for it is simply:

\[
u(k) = K_p e(k - 1)
\]

where

\[e(k - 1) = r(k - 1) - y(k - 1)\]

r is the reference input.

P-Controller is also fast in terms of the time required to converge to the steady state. The problem with P control is it is not accurate. We make it more accurate by adding a pre-compensator, but it cannot eliminate the uncontrolled disturbance input.

PI controllers have control input as the sum of proportional and integral errors, so it will always drive the steady state error to zero, but it is slower than P Controller. The control law for PI controller is:

\[
u(k) = u(k - 1) + (K_p + K_I)e(k) - K_p e(k - 1)
\]

We tested both P controller and PI controller for data source control and the results are compared in 5.2.
4.3 Controller Block Diagram and Implementation

The block diagram of P controller and PI controller for data source are shown in figure 4 and figure 5, respectively. The block diagram for queue length controller is exactly the same as the diagram of data source PI controller besides the parameters for transfer function and controller.

The implementation of controller is very easy. Most of the effort are spent getting the feedback data feed into the controller and control the sample time correctly. The controller itself only implements the control law, which is simply two multiplications and one addition.

Note that the model for queue length does not capture the fact that system can get saturated and enters a non-linear operation region. For example, the input buffer can be empty and thus not as many tuples as required by control input $u$ can be sent. Or the controller may want to send a negative number of tuples that reduces the queue length, which is not possible also. We deal with this case by making the max number of tuples it can sent to be the current queue length and the minimal number of tuples it must sent to be 50, a number that keeps TCQ happy (we suspect it is a bug in TCQ that once we stop sending tuples, TCQ will take a while to respond to next block of tuples). These artificial modifications made the controller slower, and may cause more overshoot, but as shown in the result, it still works correctly in the linear operating range of the system, while the operation out of linear range are also as expected.

5. EXPERIMENT RESULTS

5.1 Experiment Setup

We used TCQ implementation checked out from CVS on Nov. 1st, 2004. The only modification to the source code is letting it log result queue length whenever an enqueue or dequeue happens. It is configured to log only minimum number of events besides the result queue length. The size of result queue is set to about 512MB.

Other components of the system are implemented in Java. These components includes: data source that read data from a file on disk and output them through socket. TCQ input buffer runs as a separate process that takes take from data source and sent them to TCQ. There is a TCQ log monitor that monitors the TCQ log and report queue length back to the buffer controller every 2 seconds. The result from TCQ is obtained by reading output from TCQ frontend application psql by a separate process and correctness are checked there. No other known resource-hungry applications ran on those nodes while the testing was going on.

All Java processes runs on Sun Java HotSpot Client VM (build 1.4.2-04-b05, mixed mode), with three special switches -Xms512m which sets the initial heap size to be 512MB, -Xmx1024m, which set the max heap size to 1024MB, and -Xincgc, which turns on incremental garbage collection. The effect of the third argument is discussed in section 7.

5.2 An accurate data source

By adding feed back control to the data source, in both P and PI cases, it is much more accurate. As shown in figure 6 and 7, the blue line, which is the actual output data rate from the data source is always close to the red line, which is the desired data rate from data source.

An interesting thing to notice is that P controller is not exactly accurate, it is always larger than the desired value. However the PI controller is always around the red line. This effect is more clearly shown in figure 8 and 9.

The P controller cannot be accurate since there are disturbances not captured with in the model, while PI controller can always get steady state error to zero even though there exists disturbances.

The small oscillation in the figures comes from Java garbage collection. Before configuring Java garbage collection to incremental, it has the behavior of having a large drop of data rate once about 20 sec. Note that the garbage collection is expensive since data source is managing a fairly large buffer and creating and discarding string objects as it reads them off disk and send out through network.

P shows less oscillation than PI because P controller is faster in responding to a large error input.

5.3 Controlling the queue length

By regulating the free space on the result queue to 400MB, we can see from figure 10 that output buffer set its output data rate around 2000 tuples per second and the queue length is stable at around 400MB, as expected. Note that we are conservative here by allowing a large number of unused space on the result queue to allow more than enough time for the controller to overshoot without dropping tuples.

As shown in the figure, the drop in free space in queue drops at 50 sec. is due to the start effect and overshoot of controller and it becomes stable from time 50 sec to time 750 sec. Note that from 50sec to 500 sec, the source data rate is actually higher than the data source can handle, the extra tuples are queued in the input buffer. Of course, the input buffer can potentially fill up and cause tuple drops, so the buffer is better to be used as a building block for a load balancing system, as described as a future work in section 8.

Starting from time 750 sec, the input data rate is less than the max throughput of TCQ, so instead of let the control system struggle to keep the free space on the queue length (which will cause the queue length oscillate around 400MB), we specify a special condition so that the buffer controller can send out at most the number of tuples it has in the buffer. That avoids oscillation and makes sure adding the controller does not cause problems in other cases for which control is unnecessary.
Figure 4: The P Controller with pre-compensator for the data source. The reference input R is the desired data rate and output Y is the actual output data rate measured at the output point of the data source. $K_p$ is the controller and $K$ is a pre-compensator.

Figure 5: PI Controller with pre-compensator for the data source. The reference input R is the desired data rate and output Y is the actual output data rate measured at the output point of the data source. The P part and I part of controller are combined in a single transfer function. The controller parameters are estimated with the goal of making settling time to be 3 sample time periods and maximum overshoot to be 10%.
5.4 System under disturbance

Feedback control has its advantage over feed-forward control (or open-loop control) in that it is not affected by disturbances in the system as much. In this section, we verify this property.

There are many ways a computer node can fail. Since stream processing runs in a coordinated manner over multiple nodes, a failure on a node impacts the entire system. We only simulated CPU contention as disturbance, while other disturbances we want to implement are described in Appendix A.

As shown in figure 11, we started a CPU intensive process on the TCQ node, which cause a lot of contention with TCQ backend process, which is also CPU intensive. This will cause the result queue to fill up.

However, the buffer controller, almost at the same time, reduced the output data rate to around 1,500 tuples per second, which is the max throughput of the system under contention and the free space on the result queue almost has no change.

At time 480 sec, we killed the CPU intensive process, we see that the controller increases the output data rate back to normal level and the queue length still stays at the desired level.

This property is very desirable in stream processing systems since it actually allows extra time for the system to work at a degraded state correctly (i.e. not dropping data in this case). This time can be important for diagnosis of system problems.

6. DISCUSSION

In this section, we describe some practical experience we got during the design and implementation of this system.

6.1 Building system on a theoretical foundation
Figure 10: Behavior of TCQ node with PI controller on input buffer to regulate the TCQ result queue length. The top plot is the source data rate, the middle one is the output rate of TCQ input buffer (i.e. number of tuples entering the TCQ processing engine) and the bottom plot shows the free space on the result queue. This shows the success of regulating result queue length. By taking free space on TCQ result queue as feedback to the input buffer, we not only avoid the tuple drop with the same workload as shown in Figure 2, but also make TCQ work at its maximum throughput.

Figure 11: Behavior of TCQ node with PI controller, with CPU contention. The top plot is the source data rate, the middle one is the output rate of TCQ input buffer (i.e. number of tuples entering the TCQ processing engine) and the bottom plot shows the free space on the result queue. At time 180, we started a CPU intensive process on the TCQ node, where only one CPU is available. We see that the buffer output rate automatically reduced about 1/4 in order to keep the free space on the result queue to be constant.
The robustness of feedback control system is built on a sound theoretical foundation, which means we can analyze and predict the behavior of the system before implementing it. In this step, we show that it is actually necessary to do such analysis before doing a naive implementation and how bad the system can behave in a bad design of controller.

Figure 13 illustrated the instability caused by a careless implementation of controller. We started with implementing the accurate data source with feedback control. This data source is implemented as a Java class which takes in data from a in memory buffer and read desired data rate for each timestamp from a configuration file, monitors the current output rate, and use controller to calculate the actual data output rate. When implementing the buffer controller, we reused this thread and the result is as shown in figure 13.

The reason for this instability is that the data source output thread contains a transfer function (which is the close-loop transfer function of the controller shown in Figure 6). By reusing the code, the controller is placed into the queue length control loop, forming a controller as shown in Figure 12. In this case, the transfer function of the target system for the queue length controller will be the data source close-loop transfer function multiplied with the TCQ transfer function, which is not first order any longer. And a simulation with Simulink shows consistent result (oscillation) with the observed behavior.

This gives us the lesson that having the ability to analyze system behavior based on theory provides a powerful way of designing robust systems and avoid careless mistakes.

7. CONCLUSION

There are clear advantages for using control to improve the throughput and accuracy of streaming data processing systems. The introduction of control theory makes systems more robust under disturbances. It is also possible to treat complex systems as black boxes adding control theory external to the actual analysis. This makes it possible to adapt to changes for system characteristics instead of having to change the internals of a system. This paper also shows reporting system statistics adds to the improvement of the system. Implementations can be relatively easy and modular. Finally, there are theoretical guarantees that help verify the behavior of different system components.

8. FUTURE WORK

The obvious future is to use the work reported in this paper to continue progress to having a complete, controlled load balancer for general system log processing. Another investigation is to look at the sensitivity of reducing sample time in order to reduce disturbance caused by Java GC garbage collection.

Finally, it actual practice, log processing has to deal with multiple data sources, at different rates. For example, processing web queries may combine input from apache logs as well as system accounting logs. Therefore, extending these methods to controlling the scheduling of systems with multiple input streams would be both useful and challenging.

9. REFERENCES


APPENDIX

A. SIMULATING PERFORMANCE PROBLEMS IN COMPUTER SYSTEMS

Though we only implemented disturbances of CPU contention, this is potentially a list of disturbances we want to use to test our system.

CPU failure On a multi CPU job, an individual CPU may fail but the node can continue to process. If the node has N CPUs, a one CPU failure will result in the node processing at (1-1/N) normal rate. Short of turning off a node C which might be possible to do administratively it is possible to run a CPU bound task on the node in addition to its normal workload. To assure that the CPU bound job dominates a CPU, it can be made to run at a much higher priority than the other workload.

Memory failure It is possible for a section of memory to fail, but the rest of memory continues to operate. If the memory is close to fully subscribed, then the result is
Figure 12: An unstable control system, block diagram. A careless implementation that builds a feedback loop for controlling the output rate of data source inside the feedback loop of the feedback loop for queue length, which produced the result shown in Figure 13.

Figure 13: An unstable control system. A careless implementation that builds a feedback loop for controlling the output rate of data source inside the feedback loop of the feedback loop for queue length as shown in Figure 12.
that paging begins, which provides slower response to some or all of the processes on that node. To simulate the behavior, a process can be made that uses a significant amount of memory (the memory that would be equal to amount of memory that failed). An additional mode, if allowed by the operating system is to have the memory hog process pin the memory (which would be similar to having some physical memory out of service).

**Significant network traffic** On many systems, network traffic causes interrupts. Systems have to service the interrupts. It takes a relatively long time to process interrupts so if many occur, then the system becomes slow. This can be mimicked by a process that generates an interrupt for some reason (maybe using a floating-point exception).

**Memory Leaks from application or system** This is a gradually growing problem. As memory gradually reserved the system begins to page. The longer the leak goes on, the slower the system gets. The same type of memory hog program that is used for the memory failure can simulate this. This program would gradually increase its memory usage, and the system would gradually slow down. It is probable that the program will need special privileges to bypass physical and virtual limits.

**System Process Interference** System processes (or daemons) usually run at higher priority than user processes. Sometimes, configuration problems or other causes will result in the system processes running more often than usual (or needed). They take up more resources, and the lower priority processes slow down. This can be simulated with the same approach as the CPU failure noted above.

**Packets are dropped** Dropped packets limit the network transfer and are particularly disruptive for large data transfers. The dropped packets will limit (reduce) the window size for transfers, which decreases exponentially and increases linearly. This can be simulated by the receiving process manually adjusting the window size of transfers explicitly.

**Application Program failures** Typically, this will generate some type of abort or stop, which result in processing. Applications may have self correction behavior so they may restart after some interval. This can be simulated by having the application occasionally (and randomly) do long waits.