Regulating workload in J2EE Application Servers

Wei Xu Zhangxi Tan Armando Fox David Patterson
{xuw, xtan, fox, patterson}@cs.berkeley.edu

Abstract

In this project, we design and implement flow control in a J2EE application server by applying control theory and dynamic probabilistic scheduling. The goal is to regulate workload at the web front end to prevent overloading the shared database, while keeping fairness over all requests. Since workload in an enterprise application has a much larger variance in terms of resource demand, classical control theory does not work well. We supplement the feedback control with workload classification and dynamic queue scheduling, and find that correct queueing policy can simplify the control design. The experimental results show that our method effectively prevented database overloading and thus preventing deadlock without resources overprovisioning, as well as keeping client response time low.

1 Introduction

Many large software systems today are built out of black-box components. Data or requests are pushed through these components at different stages of processing. Controlling data flow in such systems is very important, but hard to tune manually. In this paper, we present a case study of using feedback control theory to control data flow in JBoss application automatically.

In a typical three tier J2EE application, the web server executes Servlets to generate response. Servlets may invoke one or more Enterprise Java Beans(EJBs), and EJBs can access database backend (Figure 1). Each tier is unaware of the capacity of other tiers. Overloading a backend tier causes problems of poor performance, even deadlock. Unfortunately, JBoss application server we used has this problem. Recovery from overloading situation is hard since front tiers, which has no knowledge of backend load, will continue to retry, causing more load.

Careful capacity planning among tiers helps prevent this problem. However, this is a time consuming task for system administrators and often requires knowledge about particular application. Due to unpredictable disturbances in the system, it is usually required to over provision resources in the system.

Our solution is to use feedback control to regulate rate at which workload enters the web server, based on the workload at the database server.

Control theory has been successfully applied to many applications in computer systems. It can be used for parameter tuning to improve system performance. The controlled configuration parameter can be resource (e.g. CPU time) allocation [7], buffer size [5], or some other configurable parameters [4].

In our previous project, we used control theory to regulate data flow. In [10], we successfully prevented stream data loss when performance disturbances causes overloading in the system. This project is more involved because enterprise applications have more heterogeneous workload that is difficult to model. HTTP requests at the web tier do not provide enough information to predict their potential resource usage. Thus, simple linear model has to include a lot of disturbances (things not captured by the model), resulting variance in output. Although more complex modeling techniques may help to solve this problem, we choose to make small modifications to the target system to keep the model simple.

The problem we are trying to solve here is similar to congestion avoidance and queueing in the networking field. Many queuing and workload classification techniques are usually used to obtain the fairness of the scheduling. Classical time-shared examples of such schemes are Virtual Clock[11], Weight Fair Queuing (WFQ)[8] and its variations such as WF2Q[1]. WFQ is the packetlized approximation of an excellent fair sharing algorithm in ideal fluid system. Hierarchical link sharing schemes provide a way to share the bandwidth resource in different levels in a hierarchical sharing tree, such as Class Based Queuing (CBQ)[6], Hierarchical Packet Fair Queuing (HPFQ)[2], and Hierarchical Fair Service Curve (HFSC)[9].

The major differences between our problem and the problems in networking are, 1) We do not have the choice of dropping a request in the middle of processing, as most of the black-box components are not designed to handle these errors; 2) It is possible to get a global view of the system

and make centralized decisions; 3) the throughput (in terms of number of requests) is much lower than networking, thus more complex queuing and classification can be used. Thus, we can do more classification and keep more state to simplify the queueing algorithms.

The queuing policy can be designed independently from congestion control. In this project, we find that certain queuing policy can help simplify the flow controller. We borrowed these techniques, but use it for a different goal, which is to simplify system model for the feedback control loop.

In our queuing policy, we classify requests by educated guesses of their usage of database. We first generate a profile offline, and find a boundary to classify the requests to be large and small, based on the database usage[1]. At run time, we guess a request to be large or small based on the URL only, and queue large and small requests separately. This is clearly an inaccurate guess, but as we have the feedback control loop, the inaccuracy can be tolerated.

The queue scheduler acts as actuator of the feedback controller, which tries to calculate a combination of small and large requests to admit into the web server. The policy favors small requests, while preventing large requests from starving. We used an algorithm called Dynamic Probabilistic Scheduling (DPS), which dynamically adjusts the scheduling priority only by monitoring the blocking probability of each type of request. This algorithm is simple and does not need manually tuned parameters.

The resulting system (feedback control + dynamic prob. scheduling) is free of database deadlock, while maintains a close-to-maximum throughput (supporting 400 simulated concurrent clients). A simple feedback controller without categorization can support up to about 300 concurrent clients without deadlock, but only when setting the maximal database access to a lower value. On the other hand, the original unmodified JBoss only provides less than half of this throughput (150 simulated concurrent client) and still deadlocks occasionally.

The rest of the paper is organized as follows: Section 2 describes the design of modules in the flow controller and the design trade offs. Section 3 discusses the experimental results. We conclude and discuss future work in Section 4.

2 Admission Control Design

In this section, we introduce the main modules that implement the admission control. Our admission control prototype is implemented in JBoss application server as four modules, the feedback rate controller, the workload classifier, probabilistic queue scheduler and instrumentation that measures workload on the database server. This system, together with the J2EE application we used, is described in Figure 1.

2.1 The controller and feedback loop

For the purposes of our control loop, our target system includes the JBoss server and the queue scheduler.

We used the simplest controller, the Integral Controller (I-controller). A controller is a mathematical function that calculates the controller input of $(k+1)$ -th interval, usually denoted $u(k)$, with an error input $e(k)$, and potentially the history of u or e. This function, or the *control law* for I-controller is:

$$u(k) = u(k-1) + K_I e(k)$$

In this application, $u(k)$ is the number of allowed HTTP request per time period. This value is sent to the queue scheduler, which pulls requests from HTTP connection queues to realize this rate. It can be understood as tuning a knob on the queue scheduler.

The controller parameter K_I is set arbitrarily to 1. We can do this because I-controller is stable no matter what K_I is. Better system identification may help to improve performance of controller, but the as this system includes complex workload and becomes unstable at boundary conditions, the system identification is hard to do, and not very useful either. In fact, setting the I-controller parameter to a conservative value is a common practise[7].

Control error $e(k)$ is calculated with the measured output $y(k)$ and a reference value $r(k)$. $r(k)$ is the max database connection allowed per time period, while y is the measured number of database connection of last period.

Setting r is a tricky process. JBoss is unstable when close to its capacity. We choose to tune this value automatically with a TCP slow-start type of experiment, which increase r until it gets overloaded. Of course, we can use another explicit control loop to tune this parameter based on other measurable database load metric. We decide to do it as a future work as it is not very relevant to the theme of this project.

Reference input r is an artificial limit imposed on system's throughput. When $e(k)$ can be large, operators of the system have to set r conservatively to avoid occasional large errors. This significantly reduces system throughput. Thus, letting e to be as close to zero as possible and has minimal variance is the first goal of the controller.

Unfortunately, we find that this simple model does not capture the disturbances caused by the unpredictability of database workload. To get a better prediction of database workload, we tried to classify the incoming workload.

Our experience shows that we can set r to approximately 90 with a simple controller with a lot of noises, while we can

[1]This process can be done online also, but we need an efficient way of tracking database usage of each requests, which does not exist in the current JBoss infrastructure.

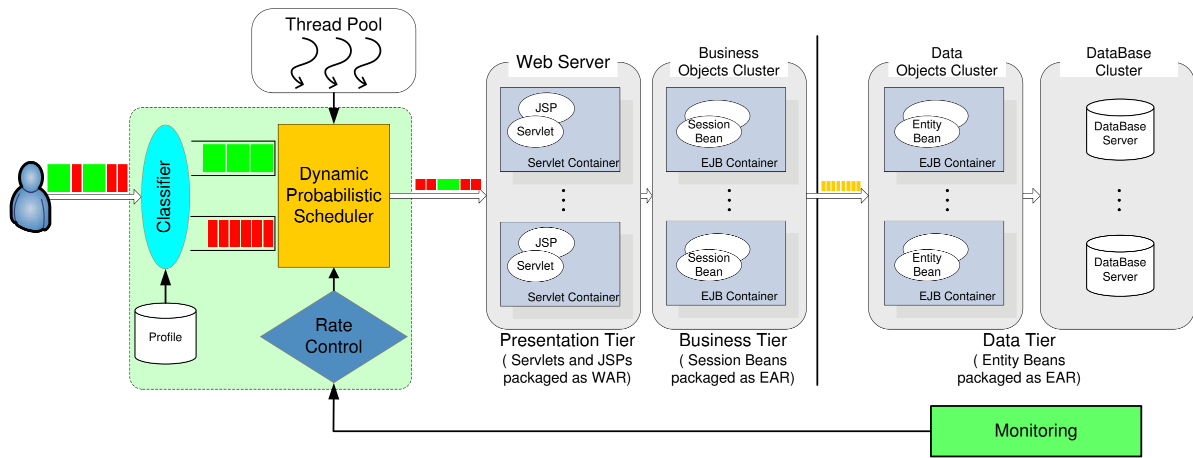

Figure 1. Overall software architecture Our admission control subsystem is implemented in JBoss application server as four modules, the feedback rate controller, the workload classifier, the queue scheduler and the instrument that measures workload on the database server. These modules implement controller, precompensator, actuator, and sensor, respectively of an abstract feedback-control loop.

set it to 120 with a controller with workload classification and separate queueing, showing a 30% increase.

2.2 Workload Classification

Although the separate queueing and workload classification techniques we use are quite similar to network queueing, they have different goals. The goal of workload classification is to get a more accurate guess of the database resource demand from a particular HTTP request to simplify the model in the control loop.

The workload classification is done in two phases. We first do an offline profiling of database usage for each URL requested. This is done by sending a sequence of low rate requests to the web server and measure the (delayed) database usage. In our future work, we want this process to be online. The process is currently not online due to the difficulty of resources accounting in JBoss, mainly resource tracking across asynchronous calls and various caching. We are investigating ways to simplify resource accounting in J2EE and similar infrastructures.

Clustering is automatically done with *k-means*, a simple statistical method. We put request into two clusters, the large requests and small requests. Notice that our measurement of database usage is only an approximation, due to uncaptured features such as parameters and caching behavior. This is enough since the feedback loop can tolerate small disturbances.

At runtime, we classify requests by their URL into large requests and small requests, based on the offline profile. This is a fast process (a single table look up). The average database usage of each class is used by the feedback controller to estimate the predicted database usage of the next period.

Workload classification naturally leads to the problem of separate queueing, since we want to avoid head blocking. Separate queueing is a policy decision. We show that certain policies can work naturally as the actuator of the feedback controller.

2.3 Dynamic Probabilistic Scheduling (DPS)

With classification, controller performance is significantly improved. However, some small requests are delayed even if they do not make any database connection. This is because the admission control is not based on the incoming rate of request, but on a separate metric of database workload.

In our policy, we want to favor small requests. There are two reasons. First, client expect simple requests to have small response times. Second, the bottleneck is database, while the web server is under-utilized. Admitting small request can increase the web server utilization and thus increase overall system throughput. Besides giving small requests high priority, we also want to avoid starvation for large requests. This is a solved problem in networking area, and we employ a dynamic probabilistic scheduling algorithm. We assign higher priority to small requests, $Priority_s > Priority_l$.

Like all probabilistic scheduling algorithms, the priority is implemented using scheduling probability (i.e. the probability that a request is scheduled upon arrival at the head of the queue.). Scheduling probability for small requests is always set to 1.0. That is, if there is any free database

connection, a small request can be served. The scheduling probability for large service $Prob_l$ is determined dynamically by our algorithm to avoid starvation of large requests.

Comparing to static probabilistic scheduling (SPS) algorithms, which requires to set blocking probability of each type of requests, DPS can theoretically guarantee service fairness between small and large requests, regardless of workload change. DPS also has performance that is close to the optimally tuned SPS and has a simple and low overhead implementation. Limited by space and the interests of this workshop, we omit the theoretical analysis of DPS in this paper.

3 Experiments

In this section, we describe our implementation of the flow control system and experimental results.

3.1 Experiment setup

We implemented our flow control system in JBoss Application server version 3.2.1. We run RUBiS [3] as our test application. RUBiS is a benchmark application for J2EE application servers. It models an online auction site with different techniques in J2EE (such as EJBs, Servlets, container managed persistency or bean managed persistency.). We used RUBiS client simulator as workload generator. The client simulator runs on 5 machines with 100 clients on each.

We set our controller parameter K_I to 1 and reference input to be 120 database requests per second as described in Section 2.1.

We describe the workload used in all experiments here (see 3). In the initial seven minutes (before the first vertical line), there are 200 concurrent clients doing normal operations (most of which are large requests after a 2 minutes ramp up time). From the 7th minute, another 100 clients that only send small request are added. At minute 10 (between the first and second vertical lines), another normal clients join (sending both large and small requests). At minute 17, 200 normal clients leave, leaving only 200 normal clients plus 100 small-request-only client until the end of the experiment.

3.2 Result

First, we want to evaluate the performance difference before and after classification. Figure 2 shows the aggregated behavior of database workload. The vertical line is the ideal case (the average database load is 120 connections per second). We see that the one with classification is very close to this ideal case. The set up with non-classification version of controller failed during the experiment since the database

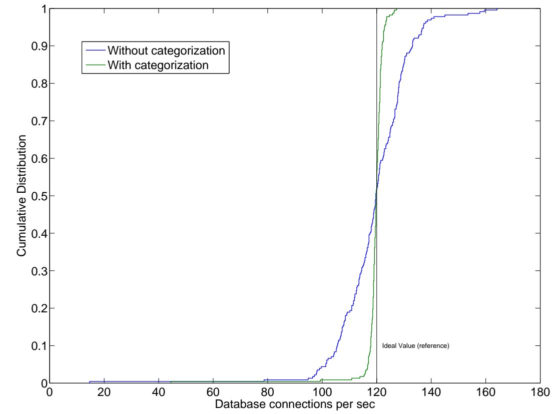

Figure 2. CDF of controller output with and without workload categorization

workload involve too much unpredicted disturbances, thus we do not show end-to-end delay figure of this type of controller.

To demonstrate the effectiveness of DPS algorithm, we compare the result with static probabilistic scheduling (SPS).

Figure 4 shows the temporal behavior of end to end response time when DPS is used for scheduling. Comparing it to Figure 3, we can see that DPS, when the database server is the bottleneck (any time except for minutes 10-17), small requests has small response time, as expected. Large requests do need to wait in a queue due to the admission control, but the response time is still reasonable (no starvation).

During minutes 10-17 (between vertical lines (2) and (3)), the workload doubled and the web server is also overloaded so database is no longer the bottleneck. In this case DPS degrades gracefully in that neither small nor large requests starve.

4 Conclusion and future work

In this paper, we implement a flow control system in JBoss application server. The aim of designing the control system is primarily for safety concern. By extending simple feedback control with request classification and separate queueing, we successfully limit the request admission rate at the web server front end and deliver robust performance when the system is overloaded. At the same time, our dynamic probabilistic scheduling assures the service fairness and better response time. The design and implementation is simple, and application independent.

For Future work, we will explore a more efficient imple-

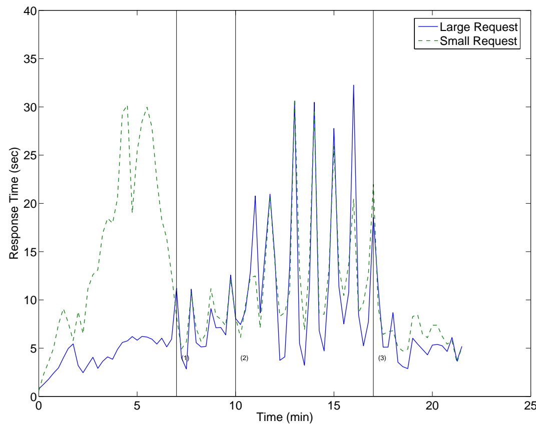

Figure 3. End-to-end response time using round-robin queue scheduling

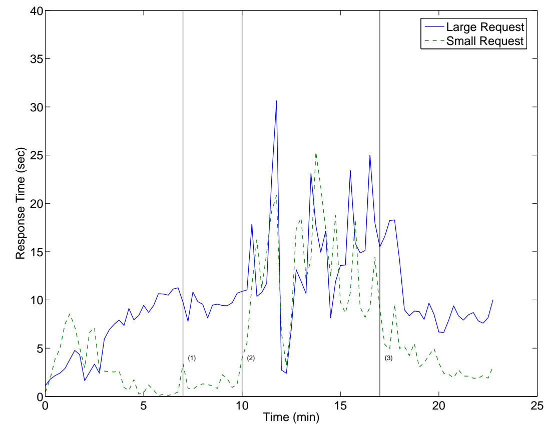

Figure 4. End-to-end response time using DPS scheduling algorithm

mentation to isolate the instrumenting and scheduling. The request classification and dynamic probabilistic scheduling can be further improved with higher accuracy and supporting multiple classes.

References

[1] Jon C. R. Bennett and Hui Zhang. WF^{2}Q: Worst-case fair weighted fair queueing. In *INFOCOM*, pages 120–128, 1996.

[2] Jon C. R. Bennett and Hui Zhang. Hierarchical packet fair queueing algorithms. *j-IEEE-TRANS-NETWORKING*, 5(5):675–689, October 1997.

[3] Emmanuel Cecchet, Julie Marguerite, and Willy Zwaenepoel. Performance and scalability of ejb applications. In *OOPSLA '02: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 246–261, New York, NY, USA, 2002. ACM Press.

[4] Yixin Diao, Neha Gandhi, Joseph L. Hellerstein, Sujay Parekh, and Dawn M. Tilbury. Using MIMO feedback control to enforce policies for interrelated metrics with application to the Apache web server. In *Proceedings of Network Operations and Management Symposium (NOMS02), 2002*, pages 219– 234. IEEE/IFIP, 2002.

[5] Yixin Diao, Joseph L. Hellerstein, Adam J. Storm, Maheswaran Surendra, Sam Lightstone, Sujay S. Parekh, and Christian Garcia-Arellano. Incorporating cost of control into the design of a load balancing controller. In *IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 376–387, 2004.

[6] Sally Floyd and Van Jacobson. Link-sharing and resource management models for packet networks. *IEEE/ACM Trans. Netw*, 3(4):365–386, 1995.

[7] Xue Liu, Xiaoyun Zhu, Sharad Singhal, and Martin Arlitt. Adaptive entitlement control to resource containers on shared servers. In *Proceedings of the Ninth IFIP/IEEE International Symposium on Integrated Network Management (IM 2005)*. IEEE, 2005.

[8] Abhay K. Parekh and Robert G. Gallager. A generalized processor sharing approach to flow control in integrated services networks: the single-node case. *j-IEEE-TRANS-NETWORKING*, 1(3):344–357, June 1993.

[9] Ion Stoica, Hui Zhang, and T. S. Eugene Ng. A hierarchical fair service curve algorithm for link-sharing, real-time and priority services. In *SIGCOMM*, pages 249–262, 1997.

[10] Wei Xu, Joseph L. Hellerstein, Bill Kramer, and David Patterson. Control considerations for scalable event processing. In *Distributed Systems Operations and Management (DSOM'05)*, 2005.

[11] L. Zhang. Virtual clock: A new traffic control algorithm for packet switching networks. In *Proc. ACM SIGCOMM '90; (Special Issue Computer Communication Review)*, pages 19–29, September 1990. Published as Proc. ACM SIGCOMM '90; (Special Issue Computer Communication Review), volume 20, number 4.