Explicitly Controlling the Fair Service for Busy Web Servers

Zhanwen Li, David Levy
School of Electrical and Information Engineering
University of Sydney
{li_zw,dlevy}@ee.usdy.edu.au

Shiping Chen, John Zic
Networking Technologies Laboratory
CSIRO Australia
{shiping.chen, john.zic}@csiro.au

Abstract

There is a growing demand for web applications to provide fair service to the highly concurrent requests. In this paper, we present an approach to addressing this requirement. Based on the Staged Event-Driven Architecture (SEDA), our design takes advantage of global control strategy to balance the loadings across the staged network, makes use of system identification to automatically model performance, and applies control theory to automatically control performance fairness. By implementing our design on a web server and evaluating the performance with unpredictable dynamic loadings, we demonstrate that our design is able to yield superior performance on fairness, showing high accuracy and good robustness.

1. Introduction

Web servers are generally needed to provide fair service for high-concurrent requests. In terms of the overheads associated with the resources contention and threading, most of the conventional thread-based concurrent models are not well suited to meet this goal. Alternative to the thread-based concurrent model, Staged Event-Driven Architecture (SEDA) [14,15] is a new software architecture to benefit the system in massive current loads and service fairness. However, SEDA cannot effectively guarantee the fairness of service quantitatively. In general, the typical justification for fairness is the current jobs receive fair service in terms of equal service time and the equal opportunity to be serviced. The percentile response time is one of the most commonly used performance metrics to evaluate the quality of fairness.

Several papers have reported the studies that make important advances towards the goal of providing explicit control on fairness. For example, Welsh [14] proposed an admission controller to manage the 90th percentile response time for each stage in SEDA. In [1], Adbelzar *et al* used feedback control for QoS (quality of service) management. In [16], Urgaonkar *et al* introduced employing scalable admission control technique for internet applications.

However, the solutions proposed in the previous work generally have a few shortcomings. Firstly, most of them made use of the arrival rate control on queuing to guarantee the fairness of service. As discussed in [1], the fairness metrics (like response time) are related to the queue length. Queue lengths may be adjusted by varying the arrival or dispatch rates. However, using the arrival rate as a mechanism to ensure fairness necessitates that any adjustments must be done at the source, which would create unnecessary rejection of traffic. Secondly, most of the current fairness control mechanisms are only developed for a single thread-based concurrency model, which is not well-suited to support high concurrent requests. Finally, a large number of the control systems adopt the fixed policy control approaches that need experience-based manual configuration, which usually cannot guarantee the quality of performance [5].

In this paper, by taking combining SEDA and Control Theory, we believe that we have a new approach to ensuring fairness under high concurrency. The SEDA architecture is deployed in our design as the groundwork to support heavy concurrent requests. Based on SEDA, our design will make use of the global control strategy to balance loading in the staged network, exploit system identification techniques to model the controlled system at real-time, and utilize adaptive control approach to implement fairness control on-the-fly.

The structure of the paper is as follows. Section 2 discusses the related work and highlights how our work differs from the existing literature. Section 3 presents the fairness control system design, including global loading control framework, self-tune stage with automatic modeling and fairness control approaches. Section 4 shows the experimental results of benchmarking our design, and Section 5 presents our thoughts on future work and conclusions.

2. Related Work

In this section, we discuss some related work on web server performance control, especially the fair service control as classified as below.

2.1 Admission Control on Queuing

The degree of fairness is generally determined by the queue lengths [1]. By controlling the queue length or the arrival rate, the response time of each accepted request can be guaranteed. Admission control approaches that are currently exploited in fairness control on software systems usually make use of round robin scheduler, or use a threshold to decide the acceptance or rejection of a request to avoid overloading. Chen *et al*. [4] proposed a dynamic weighted fair sharing scheduler to control session-based overload in web servers. The weights are adjusted to maximize the throughput objective function, partially based on session transition probabilities from one state to another, avoiding the requests overwhelming the state capacity. Similarly, Carlstrom *et al*. [3] used heuristic control approach on generalized processor sharing for scheduling requests. Based on empirical parameter configuration, Urgaonkar *et al.* [16] made use of batch processing and scalable threshold policing to handle overloads in web application servers at runtime, and demonstrated that this approach can perform well under estimated loading environments. Based on SEDA, Welsh *et al.* [14] presented a multi-stage approach to overload control based on adaptive per stage admission control. In this approach, the controller observes the staged service time and tunes request rate on each stage to attempt to meet the stage's percentile response time target.

2.2 Classical Feedback Control

Using classical feedback control theory is another approach to controlling the performance for web servers. Adbelzar *et al.* [1] adjusted control parameters based on various QoS management measures, including resource utilization, resource sharing, system loading *etc*. In [5], Diao *et al*. presented an auto-tune agent for CPU and memory utilization control by combination of automatic system modeling mechanism and LQR feedback control. Similarly, Zhou in [17] made use of the PID (proportional-integral-derivative) on queuing control, but the PID parameters configuration is based on empirical guess. Compared with other fixed policy control approaches, control theoretic approaches demonstrate advantages in their flexibility, stability, accuracy and rate of convergence. This approach is easy to use in practice, especially for software systems that need fast reaction with good robustness. However, a classic feedback control system needs the mathematical model of the target system. If the configuration of the control parameters depends on the administrator's experiences, the control quality is unreliable and non-guaranteed.

3. Fairness Control on Web Servers

As demonstrated in [14], staged event-driven architecture (SEDA) is a sound way to support highly concurrent systems. However, SEDA does not guarantee the fairness for such systems, e.g. web servers, where the client load is highly concurrent and dynamic. In this paper, we present an innovative system that explicitly manages the fairness for busy web applications with quality and quantity guaranteed.

The purpose of the fair service is to let requests equally have the expected quality of service. The distribution of the response time is an important metric to evaluate the degree of fairness. In general, the percentile response time is determined by the queue lengths, which are related to the arrival rate of the requests and the dispatch rate of the responses. A large number of studies, like the work introduced above, usually focused on using the arrival rate to control the response time. In fact, as an alternative way, resource management with respect to the dispatch rate regulation is also feasible for fairness control. In this section, we will firstly present a control approach for the throughput of the whole system, and then show how to map the percentile response time to the target throughput. Finally a control strategy for explicit fairness control by dispatch rate regulation is proposed.

3.1 Staged Event-Driven Architecture (SEDA)

SEDA partitions a complex business logic into a set of simple basic tasks in order. Each task is processed by a sequence of stages separated by event queues. Since SEDA uses non-blocking input/output (I/O) for its event-driven designs it is effectively able to provide quality-guaranteed fairness control for high-concurrent loads with less resources and contention [14]. The explicit event queues between SEDA stages act as a mechanism for controlling the flow of requests in the whole system. Under the SEDA architecture, each stage is isolated from others and is responsible only for processing a subset of requests to avoid holding resources by single request and associated thread for too long. In terms of the decoupled design and the disadvantages of the original empirical control system, in [7] we presented an innovative automatic control framework for such multiple event queue system to self optimize the system resources and performances on-line.

3.2 Automatic Control Framework on SEDA

The service rate (throughput) of the SEDA-based system is one factor that determines the latency of server requests. Queue capacity is another factor. In SEDA, loading balancing in the staged pipeline is important for the whole system performance. To meet this goal, an automatic control framework on SEDA is developed as a combination of global loading balance control system and self-tune stages. The purpose of the global control framework is to manage the overall throughput of the whole staged network at the top-level, which includes coordinating the performance of all stages in the network and balancing the loadings in the staged pipeline, as illustrated in Figure 1

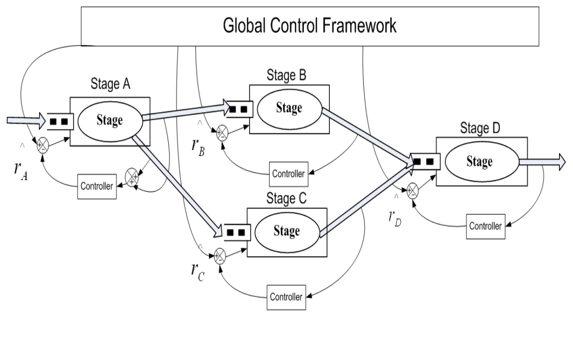

Figure 1: An Overview of the Global Control System

In this figure the broad arrows lying between the stages represent the processing path of the event requests, and the thin arrow curves represent the reference signal setup by global control mechanism and the feedback loop of the control system. $\hat{r}_A$, $\hat{r}_B$, $\hat{r}_C$, $\hat{r}_D$ are the performance targets of the stage, configured by the global control system [7].

Under the control framework, each self-controlled stage is built on the thread pool model, and will adjust its performance locally in order to meet the overall target performance. In [7] we presented a configuration law for the global workload balancing under this control framework and demonstrated that using proportional control based pre-compensator control system is able to achieve the control target with a fast convergent speed.

3.3 Automatic Control System on Stage

Each stage consists of a thread pool modular and an auto-modeling based feedback control system, as illustrated in Figure 2. In each stage, the thread pool model is the controlled target, the same as the original SEDA design. The purpose of the auto modeling mechanism is to automatically depict the flow in the stage and to optimize the controlled parameter configurations at runtime. Working together with auto-modeling, the feedback control system auto tunes the manipulated parameters to the best values by using automatic control theories, thereby guaranteeing the quality of performance.

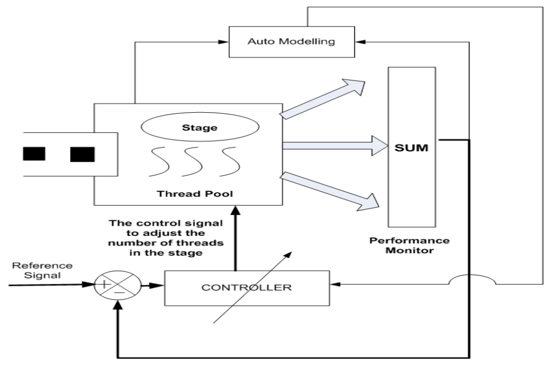

Figure 2: Self-Tune Stage

We reuse the proportional-control based pre-compensator control strategy [7] to implement the feedback control mechanism. This controller approach is validated by the results from a simulation study using Matlab [8] and a real-life implementation of a SEDA-based web server. The results are shown in Figure 3.

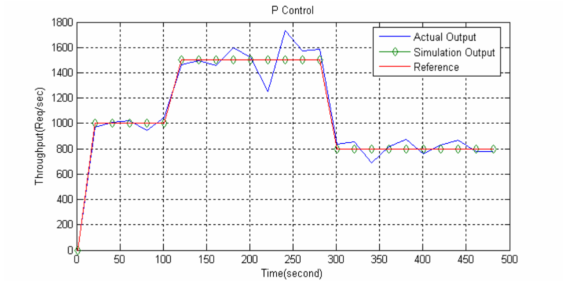

Figure 3: Performance of the Control System in Simulation and Practice

As Figure 3 shows, the simulation output closely matches the reference, which means our design is able to effectively control the system performance at runtime, with an adequate convergence rate and a very small steady-state error.

Figure 4 shown below is the result of the comparison between the heuristic control used in the original SEDA and the Proportional control strategy used in the current design.

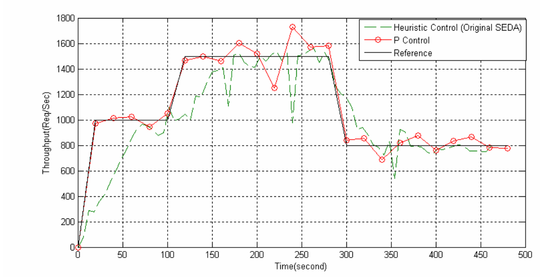

Figure 4: Comparison of the Feedback Control and Heuristic Control

The comparison results demonstrate that the performance of our control can provide a faster reaction speed, higher stability and better accuracy. The oscillations that appear in the experimental process are most probably due to the Java Virtual Machine (JVM) garbage collection [13].

In general, classical feedback control is not widely deployed in software systems because of the difficulties of manual system modeling. Combining the auto-modeling mechanism with the control system reduces the manual work in configuration management, and guarantees the values chosen for the controlled parameters are optimal and reliable.

3.4 Fairness Control

Fairness in web applications is a performance metric presented as a percentage of the requests that can get the equal quality of service. In order to guarantee the majority of requests can get the desired response time, the 90th percentile response time is chosen as the controlled target in the current research.

Based on Little's law [9], average response time can be mapped to the average number of clients via throughput as Equation (1) shows.

$$N = R.T \tag{1}$$

where N , T and R represent the average number of clients, the mean response time and the throughput respectively. Little's law implies that the average response time can be controlled by throughput. If the desired percentile response time can be mapped to the mean response time, the control approaches used in the throughput can be reused in the current design for fairness management.

Although the percentile response time does not have direct relationship to the mean response time in terms of the uncertainties of the distribution, however, regardless of the difference of the distribution, in arbitrary sample interval, the value of the percentile response time can be regarded as proportional related to the mean response time. In terms of this relationship, Equation (2) and Equation (3) are developed to estimate the desired average response time used in each sample period,

$$\tilde{R}_{estimate}(k+1) = \alpha(k+1)R_{desired}(k+1) \tag{2}$$

$$\alpha(k+1) = W\alpha(k) + (1-W)\frac{\tilde{R}(k)}{R_{percentage}(k)} \tag{3}$$

where $\tilde{R}_{estimate}$ and $R_{desired}$ respectively represent the target of the average response time obtained from estimation and the target of the percentile response time. α is a dynamic scale used to convert the target of the percentile response time to the estimated mean response time. Equation (2) converts the reference of the percentile response time to the desired average response time via scale α . Equation (3) is developed to update the α in each sample interval. At the sample interval of $k+1$, α is estimated by the most recent mean response time ($\tilde{R}(k)$) and the percentile response time ($R_{percentage}(k)$) with a weight W . Our experiment results show that, despite uncertainties of the distribution, using Equation (2) and Equation (3) is sufficient to smoothly map the percentile performance metrics to the average performance metrics with considerable accuracy at the runtime.

When system is running with a constant number of requests at any time, the clients number at time $k+1$ can be estimated by the last mean response time ($\tilde{R}(k)$) and throughput ($T(k)$), as Equation (4) shows.

$$\hat{N}(k+1) = \tilde{R}(k).T(k) \quad (4)$$

where $\hat{N}(k+1)$ is the estimated average number of clients at the interval of $k+1$ in the system.

Despite some difference between the actual number of clients in the next and current sample interval, the difference can be regarded as small enough to be neglected in a system dealing with stable loadings. Using these parameters, the desired system throughput in $k+1$ can be obtained by Equation (5)

$$Ref_{throughput}(k+1) = \frac{\hat{N}(k+1)}{R_{desired}(k+1).\alpha(k+1)} \quad (5)$$

Equation (5) maps the desired percentile response time to throughput. It means that when the average number of requests staying in the system is constant, and if the system can perform with the desired service rate, the system will ensure the fairness of service to meet the performance target. These operations are implemented with the global control mechanism. In each sample interval, the global control framework automatically adjusts the desired throughput obtained from Equation (5) on each stage. Each auto-tune stage then will use the feedback control system to convert its departure rate to this target.

Because it is a feedback control system, its response will lag changes in the workload. To address this issue, when a large number of requests are added onto the server in a small interval, the highest request rate in the recent history record will be used as the desired throughput of the stage's own performance target. This allows an increase in the convergence rate and guarantees there are sufficient resources allocated in the stage to support the increasing loading, so as to maintain a satisfactory performance. When the loading in the system becomes stable again, the global loading control mechanism will reuse Equation (2) and (3) to calculate the estimated throughput. In one sense, using a stage request rate as the desired throughput can be regarded as a special case where the service rate equals the request rate and the estimated mean response time obtained from the desired percentile response time is identical to the latest average response time.

4. Implementations and Experimental Results

In this section, we demonstrate the experimental results of putting the above theoretical designs into a SEDA based web server [15] to validate and evaluate the performance in practice. The testbed consists of one server machine (2.8 GHz Pentium 4 systems with 1.5 GB of RAM) and a client machine (2.0 GHz Pentium 4 systems with 512MB of RAM). The SEDA web server is developed with SUN JDK 1.5 running on Linux kernel v2.6.

To imitate a dynamic loading environment that is as realistic as possible, a "partly-open loop" benchmark [2] is developed to evaluate our designs. Most of the benchmarks currently in use are grouped into two catalogues. One is the "open loop" benchmarks such as httperf [6]. The benchmarks in this group send requests at a fixed rate periodically (the time of the period is also named as thinking time). The performance of this benchmark strongly depends on the configuration of the thinking time. However, the optimal configuration for this parameter is not only a matter of the benchmark itself, but is also affected by the server. In general, it is not easy to obtain a good value for the benchmark settings. Another type of the benchmark is "closed loop" model. "Clients" in this kind of benchmark will only send new request until the response of the last request has come back, like TPC-C [12] etc. Benchmarks of this type can provide constant workload on the server side. However, a pitfall of this kind of benchmark is that it cannot continue putting pressure on the server. Although it also can be used to emulate a dynamic loading environment, however, a couple of such benchmarks are needed to run alternatively during the experiment, which may consume a great deal of resources. In terms of the failures of the "open loop benchmark" and the "closed loop benchmark", "partly open loop" benchmark is proposed in [2], which is aimed to provide a more realistic dynamic loading environment by making use of a small number of the clients. The "partly open loop" system here is developed based on the SPEC 99 [11]. Whenever the response of the most recent request is not received before the expected time, the client will send a new request to the server. The "Partly open loop" benchmark simulates a more dynamic environment and keeps putting heavier load on the server with less resource consumption. The "Partly open loop" model has the advantages that it can not only guarantee a constant loading on the server, like "closed loop" model; but also behaves like "open loop" benchmark that keeps sending new requests, which enables a server to experience a more realistic loading test than conventional benchmarks.

To demonstrate our designs, we make use of the design to implement fairness control on web servers, in which the 90th percentile response time is chosen as our performance metrics and control target. This test includes two experiments. In one we have control over the 90th percentile response time in order to meet the changed dynamic target at runtime; in the second the 90th percentile response time is kept stable at the desired target when the workload is arbitrarily changed.

4.1 Trace the Dynamic 90th Percentile Response Time Target

The purpose of this experiment is to test the feasibility of the control mechanism when it is needed to control the 90th percentile response time in response to the change of the desired target at runtime. The "partly-open loop" workload generator simulates 250 clients to request the same service from the server. If the response of the last request does not come back within the expected time (1500ms here), a new request is sent to the server. The experimental result is shown in Figure 5.

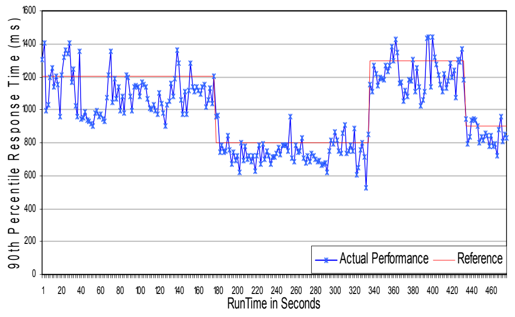

Figure 5 Adjust the 90th Response Time on-the-fly

As Figure 5 shows, whenever the performance target is changed, server adjusts the resources to follow the reference, and maintain most of the response time remaining under the target. It can be seen from the results that our approach provides a fast convergent process with high accuracy. Note that the control performance may be unsteady at times in terms of the instability of the network and the jitter of the request arrival rates, as well as the memory management used in JVM. In addition, when the control target is close to the "expected response time" of the benchmark, performance will be unstable for large variation in the request rate.

4.2 Maintain stable 90th percentile response Time

This experiment tests the performance of our design keeping constant percentile response time during unpredictable dynamic loading environment. In the experiment, four groups of clients are randomly added onto the server at different time, whose sizes are 100, 150, 100, and 50 respectively, and each client is independent on others (as Figure 7 shown). The 90th percentile response time is expected to be less than 1000ms (with 10% variance permitted). By using the above algorithm and control approaches, the performance of the system is illustrated in Figure 6. The experimental results demonstrate that even under unpredictable dynamic loading environment our mechanism is able to maintain fair service to a majority of the incoming requests.

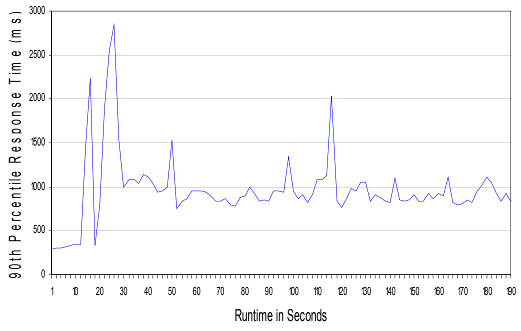

Figure 6 Maintaining Constant 90th Percentile Response Time

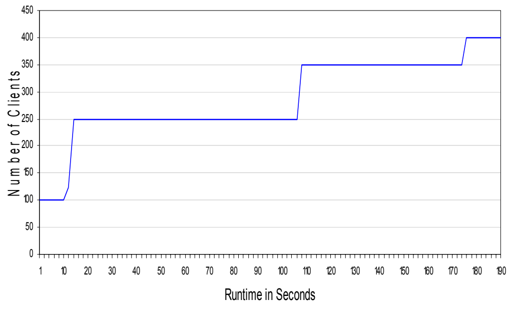

Figure 7 Dynamic Workload Placed on the Web Server

As can be seen from the results, instead of using the arrival rate control at the source, departure rate control is an alternative approach to implementing the fair service effectively for the dynamic loadings. Some unexpected spikes in the process are recorded for two reasons. The first is the variation of the request rate. In general, there is a delay in the system's reaction to the change in the working environment, and thus the system is late at adjusting the thread pool size to the correct value. Once the active number of threads is insufficient to support the requests, response time will sharply increase. Hence when a large number of new clients are added to the server, the system will take some time to settle down these new requests, similar to the performance during the interval between the 14th and 32nd seconds. Reaction delay indeed is a failure of all feedback control systems. Another reason is the periodic garbage collection in JVM, like the spike that appears at the time spot of 50th. Whenever JVM is running the garbage collection, the accepted requests are suspended, thereby accumulating a large number of requests queuing for service. The service time thus sharply increases.

This experiment also reveals the relationship of the response time, arrival rate and the dispatch rate. Figure 8 illustrates the difference of the arrival rate and dispatch rate at real-time. Referring to the response time shown in Figure 6, the experimental results demonstrate a fact that the response time is not directly determined by the number of requests accepted by the system or the service rate, but is resulted from the difference of the arrival rate and the dispatch rate. In one sense, this difference can be regarded as the total queue-fill-level in the system from the end to the front. This relationship implies that the response time should be controlled by minimizing the difference between the service rate and the arrival rate, rather than only focus on the system capacity or arrival rate.

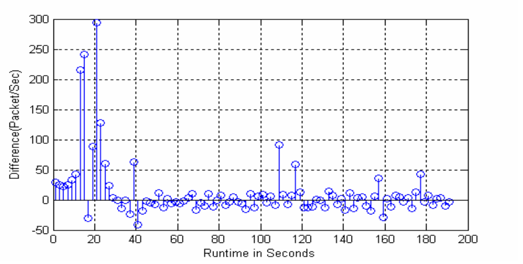

Figure 8 the Difference of the Arrival Rate and Dispatch Rate (positive value means Arrival Rate is greater than Dispatch Rate)

Figure 9 shows the distribution of the response time in this experiment. It can bee see that 90% percentile of the response time can be controlled at the points less than 1160 ms, almost meeting our control target.

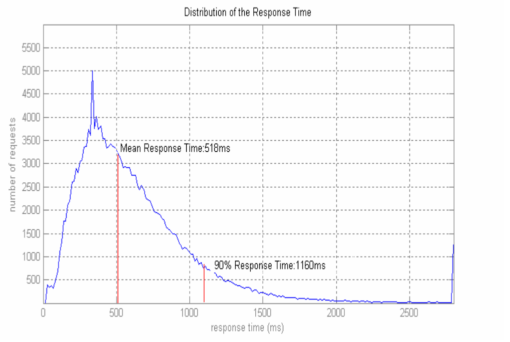

Figure 9 Distribution of the Response Time

The above experimental results definitely demonstrate that our design is able to provide explicit quantitative control on fairness for high concurrency. Even when the system runs under dynamic loading environment with uncertainties, our design is still able to effectively control the performance to meet the target and shows good robustness.

4.3 Comparison with the Original SEDA

In this evaluation, our design is implemented into a SEDA-based web server to compare with the original SEDA web server [15] that uses the admission controllers with round robins [14]. Both servers are required to provide the response time at less than 100ms to 90% of the total requests. The configuration of the parameters in the original SEDA is optimized by our long-time tests and experiences. On the client side, we make use of a set of the "open loop" request generators. Each generator emulates 100 independent clients, and every client will randomly dispatch a new request every 200~600ms. In order to build a dynamic loading environment, three groups of clients (each group has 100 clients) are added onto the server every 120 seconds during the runtime.

First of all, we compare the distribution of the response time of two servers. The results are shown as Figure 10 and Table 1.

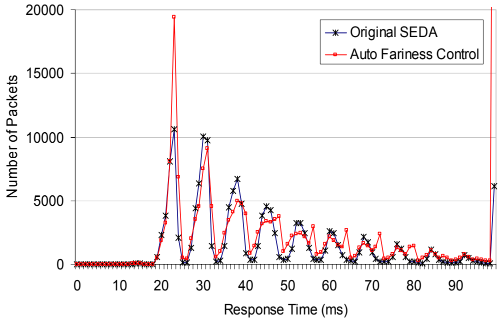

Figure 10 the Distribution of the Response Time

Table 1 the Comparison of the Fairness Performance

	Total Requests (packets)	Accepted Requests (packets)	Rejected Requests (packets)	Requests < 100ms (packets)
Original SEDA	234676	149491	85185	143362
Auto-Fairness Control	240133	240133	0	176606

The above results show the comparisons in the following aspects. Firstly, it shows the comparison on the percentage of the requests that can get the demanded response time. It can be seen from table 1 that, nearly 95.9% of the processed requests in the original SEDA can achieve the quality of service, whereas, just 73.54% meet the target in our design. Secondly, in the original SEDA, the percentage of the rejected requests reaches 36.3%, but in our mechanism the rejected rate is 0. These data indicates that, among all the client requests sent to the servers, the percentages of the requests that can get the expected response time are respectively 61.8% and 73.54% in the original SEDA and in our mechanism. Moreover, as figure 10 shows, the requests serviced by our design have an overall faster service time than the original SEDA. As a result, we argue that our approach can provide a fairer service than the original SEDA.

Secondly, we compare the throughput of two servers. The experimental results are shown in following figures.

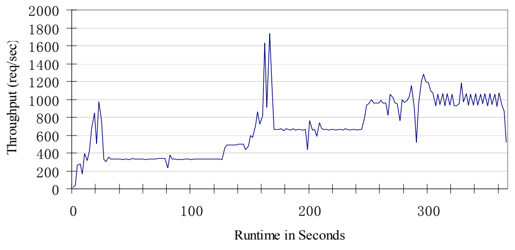

Figure 11 the Throughput of the Automatic Fairness Control

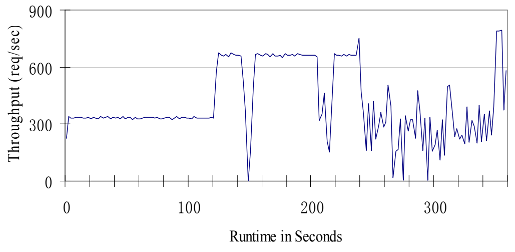

Figure 12 the Throughput of the Original SEDA under Admission Control

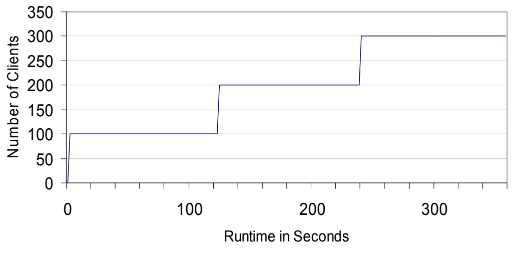

Figure 13 the Loadings at Runtime

As the experimental results show, with the number of the requests increase, our automatic fairness control mechanism is able to optimize resources to maintain a stable throughput. In contrast, under high concurrency, the original SEDA will make use of the admission control mechanism to reject the incoming requests, thereby degrading the throughput. This comparison demonstrates that our design performs more robustly than the original SEDA, and is able to guarantee the quality of the performance under unpredictable dynamic loadings.

Thirdly, as a staged network, stages are expected to perform in accord. Figure 16 and Figure 17 exhibit the comparison of the performances of the stages in the auto-tune control framework and in the original SEDA.

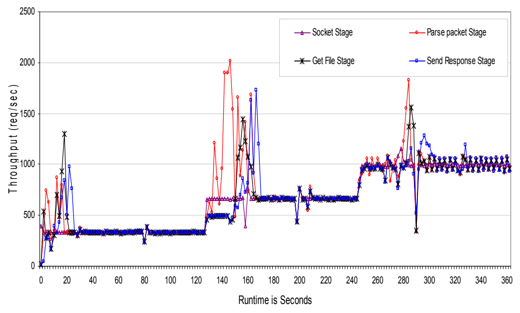

Figure 14 the Performance of the Stages under the Automatic Fairness Control Strategy

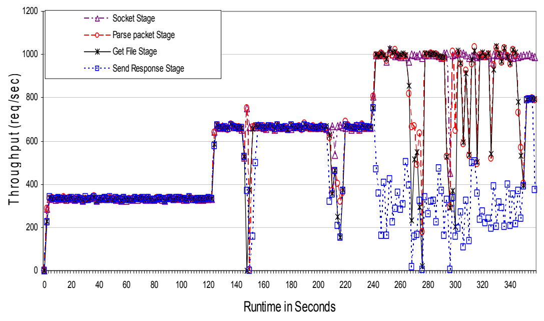

Figure 15 the Performance of the Stages in the Original SEDA

With the use of the global control framework, our auto-control system is able to balance the workload across the staged network. In contrast, under the same loading, the stages in the original SEDA perform with unequal service rates, resulting in a significant degradation of the overall performance. Therefore, our design avoids the problem that the resource is consumed by rejected requests.

Finally, our design employs the automatic control theories which implement the fairness management. Using this approach, system administrators are only needed to configure the performance target on the system. However, in terms of the fixed policy control strategy, the original SEDA is required to optimize the number of tokens and the thread pool size for each stage. This manual configuration is very time-consuming and non-quality guaranteed. In particular, when the system is running under the dynamic loading environment with uncertainties, the automatic fairness control mechanism is able to maintain a stable performance, meeting the performance target. But the performance of the original SEDA usually is degraded a lot in terms of the fixed configurations.

5. Conclusions and Future Work

This paper presented an innovative approach for explicit fairness control on web applications. Based on SEDA, the design developed here consists of a global control framework for loading balance in multi-queue the event-driven systems and the automatic modeling based high-level adaptive control stages. Both theoretical analysis and experimental results demonstrated that, by using resources optimization, our approach is effective on fairness control for web servers with superior performance even under unpredictable dynamic loading environment. Compared with heuristic control, this control strategy is able to provide fast convergence, high accuracy and reliable services. Instead of applying complicated control theory and algorithms, our work also showed that Proportional control based pre-compensation model is good enough for multiple-stage software systems performance control. This makes it feasible to build our approach into SEDA middleware and apply it for a large range of applications

Compared with the fairness of service provided by the original SEDA, our design provides a fairer and more robust service for high concurrency with no rejected requests. As a result, this work significantly improves the original SEDA in terms of service fairness and robustness.

From here, two areas of future work are particular interesting. The first is to extend our design to support differentiated services. Our current design only focuses on providing quality-guaranteed service fairness to all requests on the same service level, but is not developed for differential classes. The other is to apply our design and the approach in the large scale distributed systems.

6. Reference

[1] T.F Abdelzaher, J. A Stankovic, et al.: Feedback performance control in software services. Proc IEEE control systems magazine (2003)

[2] S. Bianca, et al.: Closed versus open system models and their impact on performance and scheduling. (2005)

[3] J. Carlstrom and R Rom, Application-aware admission control and scheduling in web servers. In IEEE Infocom 2002.

[4] H. Chen and P. Mohapatra: Session-based overload control in QoS-aware Webservers. Proc. of IEEE INFOCOM2002, pages 516-524. (2002)

[5] Y. Diao, J.L Hellerstein., et al.: Managing web server performance with autotune agents. IBM System journal Vol 42, No 1, pages 136-149. (2003)

[6] Httperf http://www.hpl.hp.com/research/linux/ httperf/docs.php

[7] Z. Li, D. Levy, S. Chen, et al.: “Auto-Tune Design and Evaluation on Staged Event-Driven Architecture” proc of MODDM '06 Melbourne (2006)

[8] Matlab http://www.mathworks.com/

[9] D.A. Menasce and F.Almeida., Capacity Planning for Web Services Metrics, Models, and Methods, Prentice Hall PTR Upper Saddle River, N.J.07458, ISBN: 0-13-065903-7. (2002)

[10] G Pacifici., W Segmuller., et al.: Managing the response time for multi-tiered web applications, IBM, Tech. Rep. RC 23651, 2005.

[11] SPECweb99 Benchmark, http://www.spec.org/web99/

[12] TPC-C: http://www.tpc.org/tpcw/

[13] Tuning Garbage Collection with the 1.4.2 Java[tm] Virtual Machine. http://java.sun.com/docs/hotspot/gc1.4.2/

[14] M. Welsh and D. Culler: Adaptive Overload Control for Busy Internet Servers. Proc. of the Fifth USENIX Symposium on Internet Technologies and Systems (2003)

[15] M. Welsh, D. Culler, et al.: SEDA:An architecture for well-conditioned scalable internet services. Proc. of the 18th ACM Symposium on Operating Systems Principles, anff, Canada. (2001)

[16] B. Urgaonkar, P. Shenoy. Cataclysm: Handling Extreme Overloads in Internet Applications. Proceedings of the Fourteenth International World Wide Web Conference (WWW 2005). pp. 740-749. Chiba, Japan.(May 2005.)

[17] X. Zhou, J Cai,, et al: Robust Application-level Approach for Responsiveness Differentiation Proc. of the 3rd International Conference on Web Services (ICWS), IEEE Computer Society, Pages 373 - 380, Orlando, (July 2005).