Chapter 1 Introduction

1.1 Problem Addressing

After decades of evolution of the Internet, services over the Internet (such as web) have become a vital resource to many people nowadays. As of 10 March 2007, it was 1.114 billion people using the Internet according to a statistic from Internet World Stats [66]. Moreover, such internet-based service as e-mail, stock trading, and on-line entertainments are often considered indispensable for both businesses and personal productivity.

With the fast-growing number of users and the increasing pace of information exchange, internet service providers are expected to provide the information with an ever-faster speed. These demands bring many big challenges to the internet server designs. One of the biggest concerns for internet service providers is the service performance of server systems when enormous unexpected concurrent client requests are added onto the systems when big events occur. During such a unexpected event, unpredictable high concurrent requests would lead to extremely overload onto the servers. Under these overload conditions, the service's response times may grow to unacceptable levels, and exhaustion of resources may cause the service to behave erratically or even crash. For example, on the day of September 11, CNN.com experienced a request load 20 times greater than the expected peak; CNN was unable to handle requests to the site for almost 3 hours though, growing the server farm size [33]. Another more recent worldwide event happened on 27 Feb 2007. In this share market's black Tuesday, the global stock market steeply plunged around 5%; many electronic stock trading sites worldwide clashed for hours because of unexpected volume surges and the heavy sell-off, which sent a big warning of how vulnerable the market structure is to systems glitches and data backlogs [49].

Provisioning satisfied QoS (quality of service) to highly concurrent requests necessitates server systems can not only deal with high concurrency, but also fairly provide the required QoS to clients. A typical justification for fairness is the current jobs receive fair service in terms of equal service time and the equal opportunity to be serviced. Common approaches to ensuring the quality of service for massive concurrent requests are to apply fixed resource limits, such as bounding the number of connections etc, or to make use of admission control over the accepted number of requests (or arrival rate) at the source [11, 12, 19, 46, 59, 55, 63, 64]. However, it is difficult to fix the resource limits for fluctuating loads; setting the limits too low underutilize the resources, while setting them too high can lead to overload regardless. As discussed in [3], a fairness of response time is strongly related to the queue length. A queue length may be adjusted by varying the arrival rate or departure rate of requests. Simply limiting the number of accepted requests necessitates any adjustments must be done at the source of a queue, which can create unnecessary rejection of traffic when underutilizing resources. In addition, a large number of the systems in current use are developed based on thread-pool model. Since the overheads associated with the resources contention and threading will result in long queuing, conventional thread-based concurrent models are not able to well support massive concurrent requests.

In this thesis, by implementing the advantages of SEDA and automatic control theory, we propose a new approach to control fairness for 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 target system at real-time, and utilize an adaptive control approach to implement fairness control on-the-fly.

The global control mechanism in the current design is used to manage the performance 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. Under the global 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. The control system in each self-controlled stage is made up of an automatic modeling mechanism and a feedback module. With the application of system identification techniques, the controller parameters in the system are optimized and configured by the automatic modeling mechanism at real-time, and the feedback control theory will provide guaranteed quality for the local performance in terms of the system model.

Based on theoretical design, our design is put into a SEDA-based web server for benchmarking and evaluation. The experiments include two cases: evaluation of the control approaches, and, the fairness control evaluated by the 90th percentile response time. Based on the experiment results, we argue that our design is able to automatically optimize resources and provide quality-guaranteed performance control for SEDA applications. Even while the system works with unpredictable dynamic loadings with high variances, it can still perform as expected, and, in practice, meeting the need of a variety of demands.

Compared with the studies conducted by other researchers, our design exhibits the following advantages. Firstly, our design is a handy and effective way to guarantee fairly the quality of service (latency here) for highly concurrent requests. Secondly, the employment of an automatic modeling mechanism and feedback control theory will significantly reduce the costs of the manual parameter configurations, and provide reliable optimal parameter settings, as well as greatly enhance the system performance in a variety of aspects, including faster convergent speed, better stability *etc*. Moreover, our design is a global control strategy for multi-queue event driven systems rather than only for the single thread pool based models. In terms of the SEDA pattern, our approach should be able to fully support the general thread pool models if required.

1.2 Dissertation Roadmap

The demand of the fair service to high-concurrent requests requires a system to support heavy loading and to ensure the quality of service to these requests. The next chapter will review the related work of fairness control over high-concurrency computing systems, which includes the reviewing on the threading system and event-driven system as well as various common-used control approaches.

The third chapter details the existing designs on SEDA. This chapter reviews the whole SEDA architecture and the system performance and management under highly concurrent requests.

The fourth chapter presents our designs to implement the fair service over SEDA. The chapter emphasizes the theoretical design to implement the fairness, which includes the global control framework, system modeling (identification), and control algorithm *etc*. The design is developed based on control theory, and makes use of simulation to demonstrate that our approaches can significantly enhance the system performance. After a brief comparison of our PID based control approach and other control strategies, our experiment results demonstrate that among the control approaches, the proportional based pre-compensator control system is the best choice for SEDA to meet the needs of performance control demands.

In the fifth chapter, we use a SEDA-based web server to validate the design by benchmarking the server performance under unpredictable dynamic loading environments on two cases. The first test uses our control approach to online adjust the system service rate to catch the desired performance target that is changed at runtime, and we use the experiment results to evaluate the performances of P, PI, PD and heuristic control. The second one makes use of our design to implement fairness control on web applications, in which the 90th percentile response time is chosen as our performance metrics and control target. This case 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. The results of the percentile response time experiments show that our design is an effective approach for fairness control on SEDA-based web applications.

Chapter 2 Related Work

There is a growing demand for web servers to provide fair service for high-concurrent requests. To be qualified to meet this requirement, a system must not only support high concurrency, but also provide quality-guaranteed fair service to requests. Before representing our design, in this chapter, we review some related work on the thread-based concurrency models and performance control mechanisms for internet servers, especially the control approaches related to fair service as classified as below.

2.1 Thread–based Concurrency Model

Thread-per-request and thread pool are two models commonly used in multi threaded programming, especially for server applications. The thread-per-request model spawns a thread for each request, and destroys the thread after finishing the request. This model is relatively easy to program, for threading allows programmers to write straight-line code and relies on the operation system to process computation by controlling the threads. However, since each accepted request occupies one thread, when the number of requests increases, it requires a large number of threads to support. Once the number of threads increases to a certain degree, it will generate a large number of overheads, severely hampering the system performance.

Thread pool is a design pattern in programming. In contrast with the thread-per-request model, a number of threads are allocated to a thread pool preliminarily. When a request arrives, the application uses a free thread in a pool to serve a client request, and returns the thread to the pool after finishing the request. Since a thread pool can reduce a large number of overheads resulting from thread creation and destruction, a thread pool shows better performance and stability than creating a new thread for each task. Generally, the maximum number of threads allowed by the thread pool is set as the thread pool size. As it is well-supported by modern programming language and can be used to efficiently handle multiple tasks at the same time, the thread pool model is deployed in a large number of popular computing system designs. [61]

However, in a thread-based concurrency model, when the number of requests is beyond the thread pool limitation, no more extra threads are allowed to be added into the thread pool. Additional connections thus cannot be accepted until threads have been released from the current request process. In a typical thread-based system, a request will occupy one thread until the end of its operation. When there are no active threads in the pool, all new incoming requests need to stay in the queue. Hence, if there are any highly concurrent requests, or if there are any requests that require a longer service time, a large number of requests will then suffer long queuing. However, most web applications like web servers are usually needed to support many thousands of highly concurrent requests, instead of only tens of requests. Obviously, the thread-based concurrency model is ill-suited to meet this goal.

2.2 Event-Driven Concurrency Model

In order to avoid the limitation of threading scheduling in thread-based systems, event-driven concurrency provides an alternative approach to offer services for high concurrency demands by making use of threads in another way.

Each client request in an event-driven system is represented as a finite state machine (FSM) or continuation, in which state stands for a set of processing steps to be performed on the request. For example, the sequential flow of processing an HTTP request by the event-driven approach can be briefly structured into the following states of accept, read, write and sent (see Figure 2-1 shown) [56]

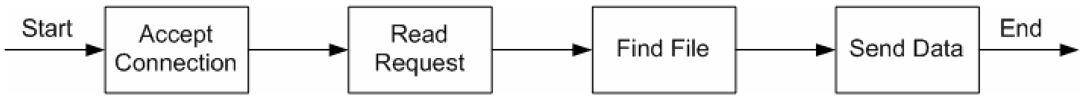


Figure 2-1 Simplified Request Processing Steps

Similarly, putting the process shown in Figure 2-1 into practice, an event-driven web server can then be modeled (Figure 2-2)

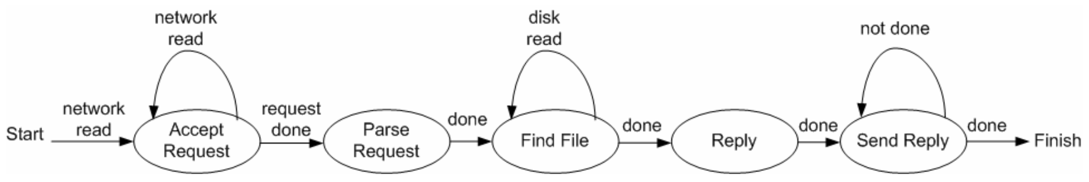


Figure 2-2 Finite state machine for a simple HTTP server request

This figure depicts a static HTTP server request as a finite state machine (FSM) as used in an event-driven system. Each state represents some aspects of request processing, and edges represent transitions between states, triggered by incoming events or the completion of the processing for a given state.

Each event processed by an FSM is typically a data structure representing a single request to the service, or may represent other information relevant to the operation of the service. Technically, the terms of the request means a request is generated by a client while other types of events may be generated by the service internally.

Unlike the above threading system design that makes use of a single thread to handle the whole processing of a client request, the event-driven mechanism deals with a request by a series of states. Each state employs at least one thread to deal with the state event operations. Hence, a thread in an event-driven system is only needed to handle the state event tasks instead of being responsible for the processing of the whole client request. In other words, a client request in a thread-based system is processed by one thread, while in an event-driven system it is processed by multiple disjoint threads in sequence. When an event request in a state is finished, the thread then dispatches it to the next state, which is handled by another thread, and the thread itself is released to service other tasks in this stage.

An event scheduler is at the heart of an event-driven system, which controls the execution of each FSM. At runtime, the scheduler always listens to the new event requests and determines which FSM should be chosen to service the event request and then dispatches the request to the correct state. To some extent, the scheduler is like a switch center for data transit among all states. When a new event arrives corresponding to some request, it is dispatched to one of the threads in an FSM by the scheduler according to the event request. As the event tasks are finished in a state, a new event request is generated and transited to another state through the scheduler. The thread in the sate is then released to operate on the next request.

Event-driven concurrency is typically more scalable than thread-driven concurrency, as the FSM state representing each request is much smaller than a thread context and many of the overheads existing in the large numbers of threads are avoided. Therefore, the processing of each state generally is very short and runs to completion due to the reduced overhead. As the number of requests increases, the server throughput increases until the pipeline fills and the bottleneck becomes saturated. If the number of requests is increased further, excess work is absorbed as a continuation in the server's event queue; this is in contrast to the thread-per-request model in which a new thread is forked for each incoming request. Hence event-driven systems are highly efficient in managing high-concurrency. Moreover, when the number of requests increases, where the number of threads in a stage is not sufficient to support the currency demands of the FSM, the new incoming requests will have to wait for the service in the event queue. In this scenario, adding some threads in the state can generally significantly improve the system throughput and reduce queuing time. [57, 61]

2.3 SEDA (Staged Event-Driven Architecture)

General Speaking, a thread-based system develops the processing logic of threads. In contrast the design of an event-driven builds the processing logic of the events. All threads in the thread-based system are placed in a pool to execute the same work, but an event-driven design allocates threads into multiple FSMs, in which each thread deals with the event tasks following the processing logic of the current state. A significant advantage of an event-driven system is that this model separates one long queue into multiple queues, so that it is able to deal with a larger number of requests at the same time, thereby supporting higher concurrency and providing fairer service.

Although an event-driven model aims to support high concurrency, most of the event-driven system performance is limited by the Input/Output mechanism [57, 60]. In order to solve this problem, a staged event-driven architecture (SEDA) is proposed [61]. By using non-blocking I/O to support an event-driven mechanism and sophisticated code modularity, the SEDA-based system significantly increases the system performance under high concurrency. In SEDA, a stage is the fundamental unit composed of a thread pool, an event queue, event handlers and various controllers. An SEDA-based application is structured as a network of stages. Each client request in SEDA is processed through a staged pipeline. Since SEDA makes use of event queues to separate stages, threads in the stage handle the stage event tasks. Therefore, even though events in some stages are blocked during processing, other stages can still work normally [61].

Theoretical proof and experiment results demonstrate that SEDA provides an effective framework for high-concurrency demands [58, 61]. However, the performance of the SEDA-based application is strongly affected by resource management. In the original SEDA designs, each stage makes use of a heuristic controller to adjust the number of threads in the thread pool. The performance of the controller is strongly determined by the controlled parameter settings. An “optimal” configuration generally depends on an experienced administrator’s good “guess”. Therefore, parameter configuration can easily result in over or under utilizing resources, and thus degrade the quality of service to requests. Moreover, the heuristic controller is developed on the basis of fixed policies, which is not suitable in a dynamic environment [59, 61]. That is to say, although SEDA is an alternative way to support high concurrency, it is not feasible to guarantee the quality of service to the requests, and cannot fairly provide the quality-guaranteed service under dynamic heavy loadings.

2.4 Admission Control on Queuing

For internet servers, the quality of fair service is strongly related to the queue lengths [3]. By controlling the queue length, the response time of each accepted request can be guaranteed. Admission control approaches are commonly exploited in providing fair service for software systems, which usually make use of a round robin scheduler, or a threshold to decide the acceptance or rejection of a request. In [63] Zhou *et al*. propose an approach called selective early request termination to prevent the system being harmed by long requests. Similarly, Blanquer *et al*. [11] exploit sliding window and selective dropping to ensure the throughput and response time for internet cluster servers. In [13] Chen *et al*. propose 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. Carlstrom *et al*. [12] use a heuristic control approach on generalized processor sharing for scheduling requests. Based on empirical parameter configuration, Urgaonkar *et al*. [55] make use of batch processing and scalable threshold policing to handle overloads in web application servers at runtime, and demonstrate that this approach can perform well under estimated loading environments. Based on SEDA, Welsh *et al*. [59] present a multi-stage approach to overload control, based on adaptive per stage admission control mechanisms. 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. This design has weaknesses in that it cannot efficiently achieve the performance target of the whole system in terms of the stage-based heuristic control algorithm, and requests may be rejected late in the processing pipeline after it has consumed a great deal of resource in upstream stages [46].

2.5 Classical Control Approaches

Using classical feedback control theory is another approach to controlling the performance for servers. Adbelzar and Lu *et al*. [1, 3, 4, 38] adjust control parameters based on various QoS management measures, including resource utilization, resource sharing, system loading etc. Diao and Hellerstein *et al*. [19, 30] present auto-tune approaches for performance and resource control by a combination of automatic system modeling mechanism and various feedback control techniques. Similarly, Zhou [64] makes use of the PID (proportional-integral-derivative) on queuing control, but the PID parameters configuration is based on an 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 control system. If the configuration of the control parameters depends on the administrator's experience, the control quality would be unreliable and non-guaranteed.

2.6 Other Control Strategies

Both PID and LQR need the mathematical model of the controlled system to develop the control mechanism, and to optimize the configurations of the controlled parameters. When the system becomes more complicated, the general system identification may not perform so well, which will limit the performance of these model-based control approaches. Alternatively, neural networks provide a new strategy to control a system without the need for system modeling. By developing multi-layer neural networks and self training, artificial neural networks can seek the optimal weights of the functions between neurons, thereby self-modeling the system. Based on the model, other control techniques can be employed to achieve the performance target. Although neural networks is a very promising way to reduce a great deal of manual effort in system modeling, it generally needs a lengthy training process to achieve a correct model. Obviously it may not be a good choice for computing systems that require fast convergent speed. In conclusion, we only can say the application of neural networks in computing systems is promising, but is still in a premature stage.

Fuzzy control is another intelligent control design attracting much attention in recent years. Fuzzy control has the advantage that, the solution to the problem can be cast in terms that human operators can understand, so that their experience can be used in the design of the controller. This makes it easier to mechanize tasks that are already successfully performed by humans. The most attractive point of fuzzy control is that this approach can make control of the systems without the need for the system's mathematical model. Unlike many other control approaches, fuzzy control provides a smooth states transition during the control process. Fuzzy control changes the system states according to the fuzzy control logic. This approach on the one hand simplifies the control design, but on the other hand it will result in slow convergence. Diao *et al.* in [20, 22] propose a fuzzy control based approach to control response time for the Apache Web Server. Although it can achieve the target performance, the convergent process is very slow.

Unlike the above passive control techniques that react to current error, model predictive control (MPC) aims to control the system prior to the error occurring, such as shown in [31, 50]. MPC relies on an empirical model of a process obtained by plant testing to predict the future behavior of dependent variables of a dynamical system based on past moves of the independent variables. Generally, prediction control is a promising approach, but it is not very flexible in its application in software systems in terms of having too many prerequisite limitations and overheads [39].

In addition, optimization search is an alternative way very commonly used to optimize the configuration for ensuring the quality of the performance. Harada *et al.* [25] propose an approach to on-line search the optimal resource allocation for fair QoS at real-time. Ling *et al.* [34] utilize the advantage of the system architecture and optimization algorithm to seek the optimal thread pool size for an Apache Server. Similarly, Liu *et al.* [36] apply Newton's law to optimize the response time online. Though optimization is a reasonable approach to achieving the best values for configuration, these approaches generally require a lot of search time and thus are not usually compatible with the systems that run in uncertain dynamic environments.

Chapter 3 Architecture for High-Concurrency: Staged Event-Driven Architecture (SEDA)

3.1 Introduction

Thread-based concurrency models are the most commonly used in current server applications. However, when the number of threads increases to a certain degree, application performance is severely degraded, thereby limiting the system's capacity to support highly concurrent requests. Alternative to the thread-based concurrency model, Staged Event-Driven Architecture (SEDA) is a new middleware architecture to support massive concurrency demands. SEDA models an application as an event-driven staged network, in which each stage is interconnected with event queues and supported by non-blocking I/O, thus avoiding the resource contentions and the scalability limits of threads. As demonstrated in [61], this design can greatly benefit the system in massive concurrent loads.

3.2 Staged Event-Driven Architecture (SEDA)

Staged event-driven architecture (SEDA) [57, 58, 59, 60] is a software architecture designed for high-concurrency. Before presenting the SEDA in depth, we here present a high-level overview of the main aspects of it.

SEDA structures an application into a network of stages, which are separated by event queues. Stages in SEDA are similar to the above states of FSM in an event driven system, the fundamental unit of event processing. Each stage consists of an event queue, a thread pool, an event handler and a variety of controllers, so that every one is capable of receiving messages, processing data, sending messages to other stages and controlling its local performance. In SEDA, each stage can be viewed as an independent little machine or actor with a distinct role or responsibility. Threads in a stage are only to process the tasks in this stage.

In SEDA, a client request is handled by multiple threads allocated in a series of stages. When the tasks of an event request in a stage are finished, a new event request is generated and passed to the next stage by a message. Communication between stages is a matter of data switching between threads. Events are oriented between stages in SEDA by an event selector, which globally listens to event requests and delivers the event tasks to the appropriate stage, similar to an event scheduler.

SEDA employs non-blocking I/O [8, 52, 57, 59] as the fundamental of the event processing, and makes use of thread pool design to drive executions in each stage. This design enables SEDA to not only provide fairness services by a small number of threads in order to deal with high concurrency, but also has the advantages of threading design, thereby significantly enhancing the system modularity and allowing the system to optimize its resources on-the-fly.

In SEDA the resource and performance can be controlled by various controllers. In the original design, SEDA makes use of heuristic control to automatically tune various resource usage parameters in the stage. Each stage is developed as a control system in which everything can be controlled by itself internally. This design is simple and has the potential to handle dynamic loadings, but its performance quality is determined by the experience-based parameter configurations, which would cost a great deal of work and limit its flexibility and performance. We will discuss the SEDA management in depth in the coming sections.

3.2.1 Fundamental Unit of SEDA: STAGE

As a fundamental unit of SEDA, a stage is developed as a combination of an incoming event queue, an event handler, a thread pool, and one or a few controllers used to adjust various controlled parameters for performance control.

Any event processed in a stage firstly enters the event queue before achieving the service. If there are threads available, they will take events out of the event queue and process the event requests immediately. Otherwise, the event will stay in the event queue until a thread is released from the current work. An event queue separates a stage from others and makes each one work as an independent thread-based system. This design not only enhances the stage code modularity, but also enables an application to inspect and manage the event requests flowing in, thereby allowing the control approaches to be performed on a per-stage basis.

Event handlers provide the core logic of execution for each stage. It decides which operations the stage should offer to the current request. An event handler in SEDA is simply a function that accepts events as input, processes events, and enqueue outgoing events onto other stages. Based on event contents, an event handler decides which operations are required to generate a correct response to the request. Event handlers are passive component designs that are invoked by runtime in response to event arrivals, and the runtime determines when and how to invoke event handlers. An event handler itself does not have direct control over threads, input queues and other aspects of resource management and scheduling. In other words, the event handler is a design used to implement the event process, but the implementation and the resource management is controlled by the runtime.

Despite the importance of the event-driven design for SEDA, threads are the basic concurrency mechanism within SEDA. As an event-driven system, SEDA is able to use a fewer number of threads than a thread-based concurrency model to support the heavier concurrency demands. The use of threads in SEDA has a number of advantages. Firstly, it releases SEDA from the constraint of the non-blocking process, allowing an event handler to block or be preempted if necessary. Secondly, by using event queues to decouple event execution, a stage can use its own threads to process the events, reducing the interaction between events. In addition, a threading mechanism enhances the flexibility of the resource management in SEDA. It is known that if some requests in a stage require long-time service, or a large number of new requests access the stage, it will significantly reduce the number of active threads, thereby resulting in the degradation of the service rate. By using a thread pool mechanism, additional threads are allowed to be added into the stage so that it can maintain sufficient active resources, avoiding performance degradation.

Stage design significantly simplifies the development of the SEDA-based application. It enables SEDA to explicitly decompose event-processing codes into stages. By having stages communicate through an explicit event-passing interface across bounded queues, the problem of managing the control structure of a complex event-driven application is greatly reduced. Moreover, in this model, composition is accomplished through asynchronous event exchange, rather than through the more traditional approach of function call APIs. SEDA thereby can be modeled as a network of event pipelines.

3.2.2 Staged Network

SEDA structures a system as a staged network, like a flow system. Stages in the network are connected with event queues; each stage works like a node in a flow pipeline, to control and process events. In some sense, a staged pipeline stands for a process generating a response to a user request. Experiment results shown in [61] demonstrate that this design is able to support high-concurrency demands with a small number of threads. By dynamic self-control design, the size of the thread pool in each stage can be adjusted at runtime. Thereby it can take the advantage of the resource control to reduce or even completely remove the bottleneck stage effects in the processing pipeline.

Staged pipelining provides an efficient approach to controlling a SEDA-based system. In thread-based concurrency systems such as the thread-per-request and thread pool models, a user request is handled by a thread. Any unexpected performance or changes on the processing would affect the whole process. In contrast, by easy event-acceptance or ignorance, a staged pipeline design can isolate a stage from others. Therefore, when a stage is blocked or is upgrading, other stages can work as usual. Similarly this structure can also prevent error propagation.

In SEDA, concurrency is managed within each stage, scheduling is performed by the underlying thread system, and resource bottlenecks are managed through the resource controller developed for each stage. The SEDA-based system not only inherits the advantages of an event-driven system, but also combines the benefits of threading concurrency. In particular, the outstanding modularity design significantly enhances the system performance to meet various dynamic requirements.

A number of experiment results are provided in [61] which demonstrate that the staged network design greatly benefits the system design and maintenance. However the staged network requires that the control strategy is not only able to correctly control the performance of each stage, but also to coordinate all stages in the processing pipeline. If the workload in the pipeline is not balanced in each stage, system performance will be limited by the bottleneck stage. We know that the goal of the resource control in SEDA is to optimize the resource allocation in each stage, so that the system can utilize the least amount of resource to support the desired workload. Therefore, a global control strategy is needed to manage stages in the processing pipeline, balancing workloads of each stage, thereby optimizing system resources and maximizing the system's capacity. In the next section, we will discuss the control and management of SEDA in detail.

In a comparison with Apache web server [6], it is demonstrated that SEDA is able to provide a fairness service for high concurrency requests with a much lower resource consumption [58, 61]. In contrast, Apache is good at dealing with a small number of requests and offers a higher service rate. In fact, the thread-based concurrency model in some sense can be regarded as a special case of SEDA. Similar to a stage in SEDA, a thread-based concurrency system is structured by event queues, a thread pool and event handlers. Another major difference between the two is that the thread pool model deals with the whole client request in a thread pool rather than a part of the request. An approach which is able to control the performance of a stage will have great potential for extension to support thread-based concurrency systems if required.

3.3 Performance Control on SEDA

The original SEDA provides resource control and overload control [58, 59, 61]. Both resource control and overload control are implemented in each stage. By monitoring the queue length and the service rate, the resource controller employs the heuristic control approach to adjust the number of threads in each stage. This approach is easy to use and has the potential to support the dynamic loading demands. An overload controller makes use of the admission control at the source of the event queue to control the number of requests accepted by the system. Whenever the percentile response time of the stage is over the desired target, new requests are rejected, thereby guaranteeing that the 90th percentile response time in the stage is under the performance target. This approach is also extended to perform class-based service differentiation.

The most apparent advantage of the existing control designs on SEDA is that it is easy to utilize in practice to solve realistic problems. However, the performance of these control strategies is strongly determined by the controller parameter configurations. In the original SEDA, the guarantee of the quality of the system performance requires that a set of multiple relevant parameters such as threshold and thread pool size are correctly set by administrators for a number of stages. This is very time consuming and non-quality-guaranteed. An 'optimal' configuration usually depends on an experienced administrator's good guess. Therefore, the manual parameter configuration approach easily results in, over or under, utilizing resources. In addition, the control policies used in the original SEDA are fixed, which is not very suitable in some dynamic environments. Furthermore, the designs of the original SEDA are only developed for local control on a stage, but do not support the overall performance at a high-level such as for a global control strategy. In other words, SEDA provides a new architecture that has the potential to support high concurrency, but is still unable to guarantee the quality of performance of the system, and thus cannot ensure the quality of service for the dynamic concurrent requests as well.

Chapter 4 Quality-Guaranteed Fair Service on SEDA

4.1 Introduction

As discussed in the last chapter, although SEDA provides a new software architecture to support high concurrency, its control mechanism is not good at guaranteeing fairly the quality of service for highly concurrent requests. Various performance metrics have been studied in the context of fairness of service, including resource sharing [12, 24, 47], differentiated service metrics [59, 55, 64, 65], response time target [46, 64]. In this thesis, we focus on providing the jobs equal service time and equal opportunity to be serviced, and use the 90th-percentile response time as a realistic and intuitive measure of client-perceived system performance. In contrast to average response time or maximum response time, the metric of 90th –percentile response time represents the response time of the 90% accepted requests, which benefits the representation of the distribution of the response time and the fairness of service.

In order to guarantee the 90% response time at less than the target, the arrival rate of the requests and the departure rate of the responses need to be balanced, so that the queuing time would not be so long as to degrade the performance. A large number of studies, like the work introduced above, usually focus on using the arrival rate to control the response time. In fact, as an alternative way, resource management with respect to the departure rate regulation is also feasible and necessary for controlling the fairness. In this research, a new control system for ensuring fairness of service is developed. The design uses the advantages of the adaptive control mechanism to optimize the system resource at real-time, and makes use of the selective early request termination [11, 63] at the source to filter long-queuing requests, so that the system can provide the required quality of service to the maximum number of requests. This chapter will firstly present the control mechanisms, including the global control framework and the self-tune stage, and then show the algorithm used to implement the control functions. Finally we will make a brief introduction of other commonly used control approaches in computing systems.

4.2 Control Framework for SEDA-based Fair Service

The control system developed to implement the SEDA-based fair service aims to provide the desired quality of service to the maximum number of requests with the least resource consumption. The idea behind the design is that we firstly decide the number of requests that are allowed to stay in the system by mapping the target percentile response time to the demanded throughput, and then employ the automatic resource management to guarantee the quality of service needed by the requests.

Unlike traditional thread-based concurrency models, the overall performance (such as response time and throughput) of the whole SEDA-based application is determined by the performance of each relevant stage in the network. If each stage has sufficient resources to guarantee its desired local performance, the required performance of the whole system can be achieved.

In order to automatically control the number of requests which stay in the whole system and to guarantee that the requests can receive the required fair service, the control system is developed as a combination of a global control framework and a set of self-control stages. The global control framework manages the performance 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. 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. Compared with admission control and heuristic control, this innovative control approach can significantly improve the system performance in a variety of aspects, including flexibility, convergent speed, stability, and so on. Moreover, it can also greatly reduce the manual work on system configuration for system administrators.

4.2.1 Global Control Framework

Theoretically, the maximum throughput of a staged pipeline is limited by the service rate of the bottleneck stage, whereas some stages placed in the front of the bottleneck may provide higher throughput than the bottleneck stage. Therefore, a global control framework (as illustrated in Figure 4-1) is used to configure the performance target on each self-tuned stage according to the demand of the overall performance of the whole system.

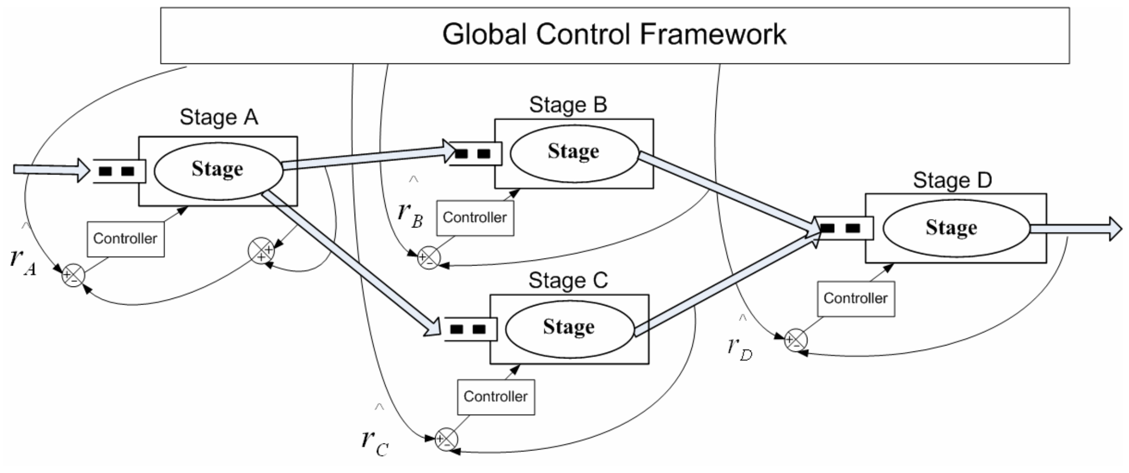


Figure 4-1 Global Control Framework

In this figure, the board 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 framework 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 each stage, configured by the global control framework. Under the SEDA-based control framework, each stage makes use of automatic controllers to adjust the stage's own resources to achieve the performance target.

The global control framework is a mechanism that controls the overall performance of the whole staged network at the top-level. It coordinates the performance of all stages in the SEDA architecture and balances the loadings in the staged pipeline. As each self-tune stage achieves the local performance target, the desired performance of the whole system is obtained.

The principle of deciding the settings in the global control model is that the workload on every stage placed in serial is expected to be the same, i.e. the maximum workload that the system expects to support, and the stages working in parallel should share the workload in a specified proportion. For example, when a SEDA application is structured (as Figure 4-1), the desired workload for each stage should be configured to hold the following relationship:

$$\tilde{T}_A = \tilde{T}_B + \tilde{T}_C = \tilde{T}_D$$

where $\tilde{T}$ represents the reference workload for the specific stage A, B, C and D. We thus define the configuration law mathematically as Equation (1) depicts.

$$\begin{gathered}
\tilde{T}_{serial} = \tilde{T}_{paraA} + \tilde{T}_{paraB} + \cdots + \tilde{T}_{paraN} \\
\tilde{T}_{paraA} = \alpha_1 \tilde{T}_{serial}, \tilde{T}_{paraB} = \alpha_2 \tilde{T}_{serial}, \ldots, \tilde{T}_{paraN} = \alpha_n \tilde{T}_{serial} \\
\alpha_1 + \alpha_2 + \cdots + \alpha_n = 1
\end{gathered} \tag{1}$$

where $\tilde{T}_{serial}$ and $\tilde{T}_{paraA}$ respectively represent the throughput target of the stage placed in serial and in parallel. α_n is the scalar, which means the proportion of the total throughput used in this stage. When each stage placed in series has the same throughput, there is no bottleneck existing in the staged pipeline.

4.2.2 Self-Tune Stage Model

In order to balance the loading in the staged network, every stage is expected to have sufficient resources to achieve its local performance target, avoiding over or under utilizing resources. As introduced above, each stage in SEDA is separated by event queues, working as an independent thread-based system. When an event request goes into a stage, it firstly enters the event queue, waiting for the active thread. When a thread is available, the request is executed following the processing logic. If all tasks of the request in the current stage are finished, a new event request or a client request response is generated and dispatched, and the thread is released to service the next request. The thread pool design allows a stage to adjust the number of threads on the fly, thereby achieving the expected performance. Before resource utilization reaches the system's maximum capacity, adding extra threads in a stage can increase the system efficiency and enhance the system capability to service more requests. Therefore, under a global control framework, a self-tune stage is modeled (as Figure 4-2 illustrates).

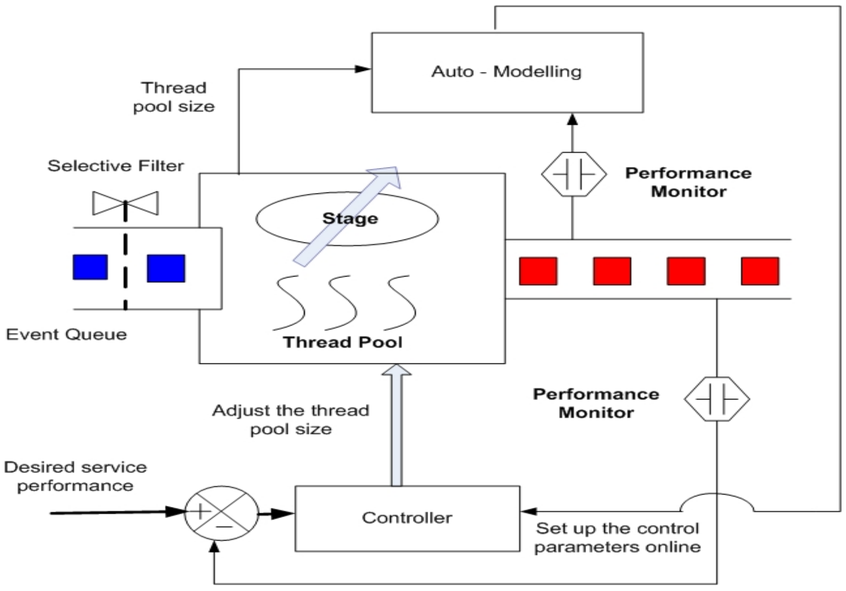


Figure 4-2 Self-Tune Stage Model

As Figure 4-2 shows, the self-tune stage consists of a thread pool modular, an auto-modeling based feedback control system and an admission controller at queuing. An event-driven stage consists of an event queue, an event handler and a thread pool. This component is a threading model used to handle the event tasks. An auto modeling mechanism automatically depicts the flow in the stage and sets up the parameters of the controller at runtime. Working together with auto-modeling, the feedback control system periodically auto tunes the number of threads to the best value by using control theories, thereby guaranteeing sufficient resources to support the desired performance. In addition, the admission control placed on the queue aims to select and terminate the long-time queuing requests. The purpose of combining these control mechanisms is to regulate the arrival and departure rate of a stage, so as to control the number of requests and ensure provision of the expected quality of service to the maximum number of requests. In the current design, we choose a selective dropper (to be discussed below) to implement the admission control mechanism.

The auto-tune stage is a fundamental unit to ensure fair service in SEDA, which benefits the system performance in a variety of aspects. Firstly, under a global control framework, the self-tune stage can automatically balance the number of requests in each stage in the staged pipeline and guarantee fairness of service to meet the requirements. Secondly, by using an automatic model mechanism, the system performance control no longer relies on experience-based manual parameter configurations, which significantly reduces the cost and provides more reliable settings for the control parameters. Thirdly, based on the mathematical model, the control theory based system design can provide faster convergent speed, better robustness and quality-guaranteed performance for SEDA-based systems. Finally, the automatic control design enables the system to achieve the desired performance with the least resource consumption. Experiment results demonstrate that this design is able to provide expected performance for the system running under an unpredictable dynamic loading environment. The details of the stage design and the control algorithms will be introduced in the sections below.

4.2.3 Early Selective Dropping

Most of the time spent by a request sitting in a server system is in queuing [39, 41, 42]. The function of selective dropping aims to discard the requests suffering such a long queuing that the deadline of their service cannot be met, thereby enhancing the quality of performance. To a certain extent, it can also avoid overwhelming requests on the server. In order to avoid the delay in receiving feedback of the response time, our implementation (Figure 4-3) makes use of the queuing time to decide whether the request is dropped or not. If the queuing time of the request is over the threshold, the request will be filtered.

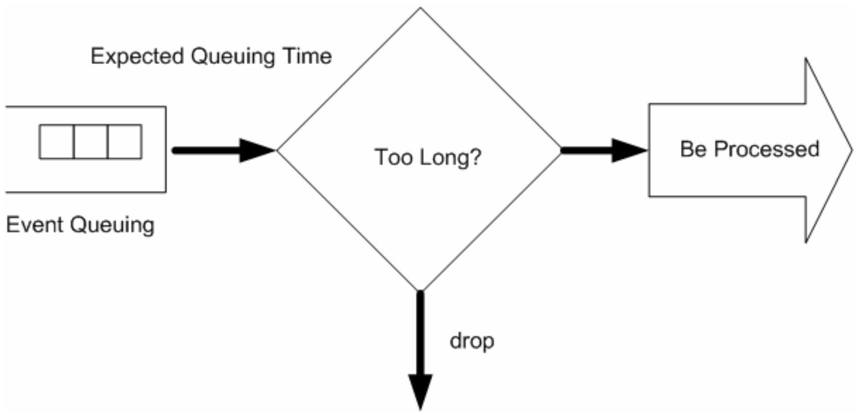


Figure 4-3 Selective Dropper

The selective dropping mechanism leverages the queuing inside SEDA to absorb safe peaks of traffic, thereby guaranteeing the response time by meeting the service rate that can be handled by the system with the current configuration. Because of the staged structure, the selective dropper is only placed in the front stage in order to reduce the resource consumed by the long time requests. In the current design, we develop a simple selective dropper. A more sophisticated selective dropper with a dynamic threshold which is varied depending on the departure rate and queuing length is under study, aimed at improving the system performance under overloading.

4.3 Control Algorithms for Fair Service

The algorithm that ensures fair service for SEDA consists of a mapping from the percentile response time to throughput and the automatic control theory for resource management. The mapping algorithm is implemented in a global control framework and the resource management is executed in each stage. In each sample interval, the global control mechanism will update the target throughput on each stage according to the distribution of the response time and the current desired percentile response time. The stage then will execute the control mechanism to adjust the resource whenever it receives a new target throughput, so that a stage can control the departure rate in quantity. In this section, we will firstly introduce the mapping algorithm between the departure rate and the desired percentile response time, and then present the control theory based resource management for throughput control.

4.3.1 Mapping the Percentile Response Time to Throughput

The basis for our fairness control strategy is that when the system resource required by the requests can be provided by the system, the departure rate controller will adjust the thread pool size to increase the processing rate for the increasing loading, so as to provide the desired response time to the incoming requests. Contrarily, when requests overwhelm the system, this controller will maintain the throughput at a high level and work together with the admission control mechanism in order to prevent overloading and guarantee the accepted requests can receive the expected quality of service.

In detail, according to the difference of the loading added onto the system, the control strategies over the departure rate are classified into two groups by our departure rate controller. Firstly, when the number of requests in the system is not greatly changed, it is presumed that the web service provider works with a constant number of clients. The desired departure rate is estimated by Equations 2 to 5, which are derived from Little's law [41], and the thread pool size of this stage is adjusted to meet the requirement of this target throughput. In opposition to that, if the loading in the server varies at runtime, then the arrival rate will be taken as the target output of the departure rate controller in order to maintain the system running at steady-state so that the arrival rate equals the departure rate. In this scenario, the selective dropping controller and heuristic control are used together with the departure rate controller to enhance the performance of the quality-guaranteed fair service.

It is well known that Little's law shows the relationship of the response time to the number of requests. This relationship implies that the average response time can be controlled by throughput when the number of requests is known. 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 a direct relationship to the mean response time in terms of the uncertainties of the distribution, the value of the percentile response time can be regarded as proportional to the mean response time in an arbitrary sample interval regardless of the difference of its distribution. In terms of this relationship, Equation (2) and Equation (3) are developed to estimate the desired average response time used in each sample period,

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

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

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 response time to the average value with considerable accuracy at the runtime.

When a 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.

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

Where $\hat{N}(k+1)$ is the estimated average number of clients at the interval $k+1$ in the system. Despite some difference between the actual number of clients in the next and the 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).

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

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.

In order to avoid over utilizing the system resource, as requests overwhelm the server, our design uses the combination of the heuristic control and departure rate controller to adjust the thread pool size and adopt admission control to control the request rate. The main function of the heuristic control is to fine-tune the number of threads in a stage in order to maintain a stable and maximum throughput. At the source of a queue, the accepted rate is managed by an early selective dropper to filter the long queuing requests, so that the system resource is not used by the long request and the demanded quality of service can be maintained during overloading.

4.3.2 Algorithm for Throughput Control

The above mapping from the response time to throughput is executed in the global control mechanism. Under the global control framework, each stage is developed as a feedback control system. In general, a feedback control system is structured as a feedback control loop (as illustrated by Figure 4-4):

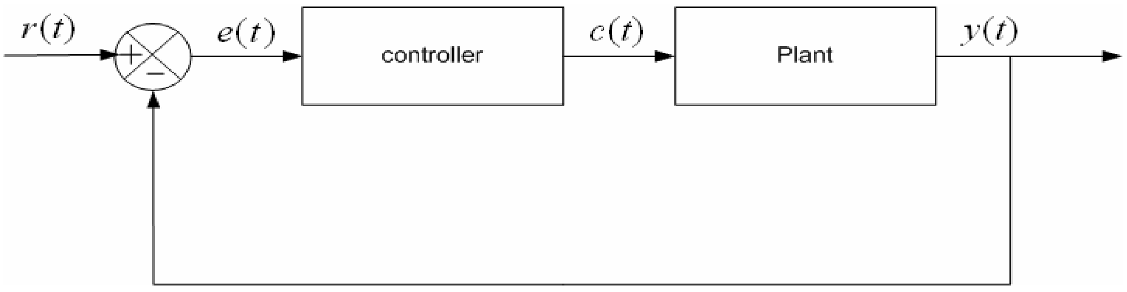


Figure 4-4 Feedback Control Model

In each sample period, output $y(t)$ is sampled and fedback to compare with the reference signal $r(t)$. The difference between them is the error, represented by $e(t)$. The controller makes use of this difference as the input to adjust the control signal for the Plant, according to the control algorithms. Since the output of the controlled system ($y(t)$) is controlled by $c(t)$, when a controller can provide the correct controlled signals to the plant, it can enable the target system to generate the desired output. When a new output $y(t)$ is produced, it is fedback to compare with the reference signal in the next sample period, repeating the above process. In this design, the controlled target is the mathematical model of the stage thread pool, represented by the mathematical relationship lying between the throughput and the thread pool size. $y(t)$ and $c(t)$ respectively denote the throughput and the thread pool size, and $r(t)$ is the desired throughput.

4.3.3 Automatic Modeling

The thread pool based stage is the control target (Plant) in the feedback control system. Modeling the stage is the most important step to build up a good feedback control system for SEDA to achieve the desired quality of performance. The purpose of the system modeling is to seek a mathematical relationship between the controlled parameters and the manipulated parameters. Here, what we need is the relationship between the thread pool size and throughput. Such physically controlled systems as mechanical or electrical systems can be modeled based on the physical laws. Unfortunately, unlike the physical system, there are no direct relationships lying between the system performance and the system resources in software systems. A computing system is like a black box that only offers a little knowledge of the controlled input and output, which limits the application of other system identification techniques. For these reasons, statistical techniques are chosen in the current design to infer the relationship between inputs and outputs. Statistical approaches have considerable appeal that can greatly reduce the information required to model the construction, instead of using detailed knowledge of the target system [30]. Among a great number of system modeling techniques, black-box system identification technique is one of the most commonly used statistical approaches to modeling the target system. In general, a computing system input-output relationship can be represented as an ARX model by a linear difference equation (Equation (6)) [30, 35, 43].

$$y(t)+a_1y(t-1)+\cdots+a_{n_a}y(t-n_a)=b_1u(t-1)+\cdots+b_{n_b}u(t-n_b)+e(t) \tag{6}$$

where $y(t)$ and $u(t)$ respectively denote the output and input at time t, and the set of a and b are adjustable parameters. A pragmatic and useful way to see Equation (6) is to view it as a way of determining the next output value given previous observations.

Equation (6) was widely used in previous designs [35]. In these designs, the thread pool size was taken as the manipulated variable $u(t)$, and the performance metrics as the controlled target $y(t)$. Based on Equation (6), the thread-based systems in the above cases were modeled as Equation (7) (such as Figure 4-5 illustrated).

$$y(k)=ay(k-1)+bu(k-1) \tag{7}$$

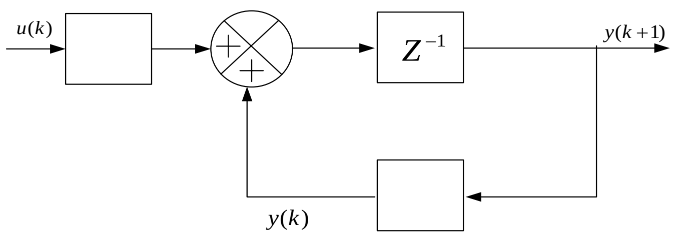


Figure 4-5 Traditional Controlled System Model

In Equation (7), $y(k)$ and $u(k)$ denote the controlled performance target and the number of threads deviating from the initial setting respectively.

Although each stage in SEDA can be regarded as a thread-based control system, for each stage, as the number of threads in the stage changes, the stage throughput will increase or decrease immediately. It means the current stage service rate is determined by the current number of active threads in the stage thread pool instead of relying on the above relation (Equation (7)). The previous system model apparently is not well-suited for our current design. Therefore we develop Equation (8) to model our system (as shown in Figure 4-6).

$$y(k) = ay(k-1) + bu(k) \qquad \textbf{(8)}$$

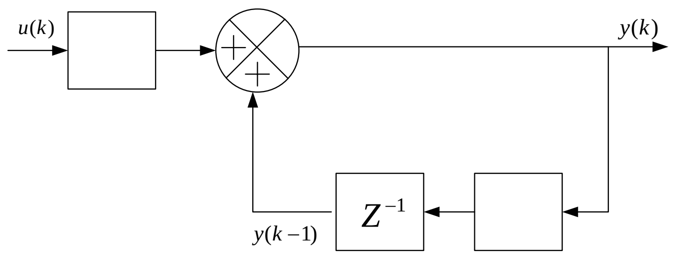


Figure 4-6 Stage Thread Pool Model

where $y(k)$ and $u(k)$ respectively represent the stage throughput and the number of threads changed in the kth sampled interval in the stage. The relationship between the system performance and the number of threads is in fact nonlinear, but using the linearized relationship of Equation (8) is sufficient to control the stage system before the number of threads reaches a certain number. This argument is to be validated by the following experiments.

Since a stage can be modeled as a single-input-single-output (SISO) linear model, the least mean square (LMS) is chosen to estimate the values for a and b in Equation (8). LMS is one of the least squares techniques for mathematical optimization. When given a series of measured data, the least squares method attempts to find a function which closely approximates the data, and tries to minimize the sum of the squares of the ordinate difference between the points generated by the function and the corresponding points in the data. Specifically, when the number of measured data is one and the gradient descent method is used to minimize the squared residual, it is called least mean squares (LMS). LMS is known to minimize the expectation of the squared residual, with the smallest operations (per iteration) [30, 35].

In the current system modeling, LMS estimates the constants of a and b in Equation (8) for minimizing the difference between the estimated and actual output.

We denote the estimated output value by $\tilde{y}(k)$. That is

$$\tilde{y}(k) = ay(k-1) + bu(k) \tag{9}$$

The k th residual then is: $e(k) = y(k) - \tilde{y}(k)$. LMS aims to choose a and b so as to minimize the sum of the squared errors. In other words, it is to minimize the function:

$$J(a,b) = \sum_{k=2}^{N} e^2(k) = \sum_{k=2}^{N} [y(k) - ay(k-1) - bu(k)]^2 \tag{10}$$

The values of a and b that minimize $J(a,b)$ can be found by taking partial derivatives and setting them to zero (Equation (11) and Equation (12)).

$$\frac{\partial}{\partial a} J(a,b) = -2\sum_{k=2}^{N} y(k-1)[y(k) - ay(k-1) - bu(k)] = 0 \tag{11}$$

$$\frac{\partial}{\partial b} J(a,b) = -2\sum_{k=2}^{N} u(k)[y(k) - ay(k-1) - bu(k)] = 0 \tag{12}$$

For convenience of notation, define the following quantities

$$S_1 = \sum_{k=2}^{N} y(k-1)y(k-1) \qquad \textbf{(13)}$$

$$S_2 = \sum_{k=2}^{N} y(k-1)u(k) \qquad \textbf{(14)}$$

$$S_3 = \sum_{k=2}^{N} u(k)u(k) \qquad \textbf{(15)}$$

$$S_4 = \sum_{k=2}^{N} y(k-1)y(k) \qquad \textbf{(16)}$$

$$S_5 = \sum_{k=2}^{N} u(k)y(k) \qquad \textbf{(17)}$$

So we can achieve the optimal values of a and b .

$$a = \frac{S_2S_5 - S_3S_4}{S_2{}^2 - S_1S_3} \qquad \textbf{(18)}$$

$$b = \frac{S_2S_4 - S_1S_5}{S_2{}^2 - S_1S_3} \qquad \textbf{(19)}$$

Here, a SEDA-based web server (Haboob [58]) is used to test and validate our design. In Figure 4-7, we show a comparison of the actual and the estimated outputs of Haboob and the above system model. In this diagram, the horizontal and the vertical axes respectively represent the number of threads in a stage thread pool and the throughput of the stage.

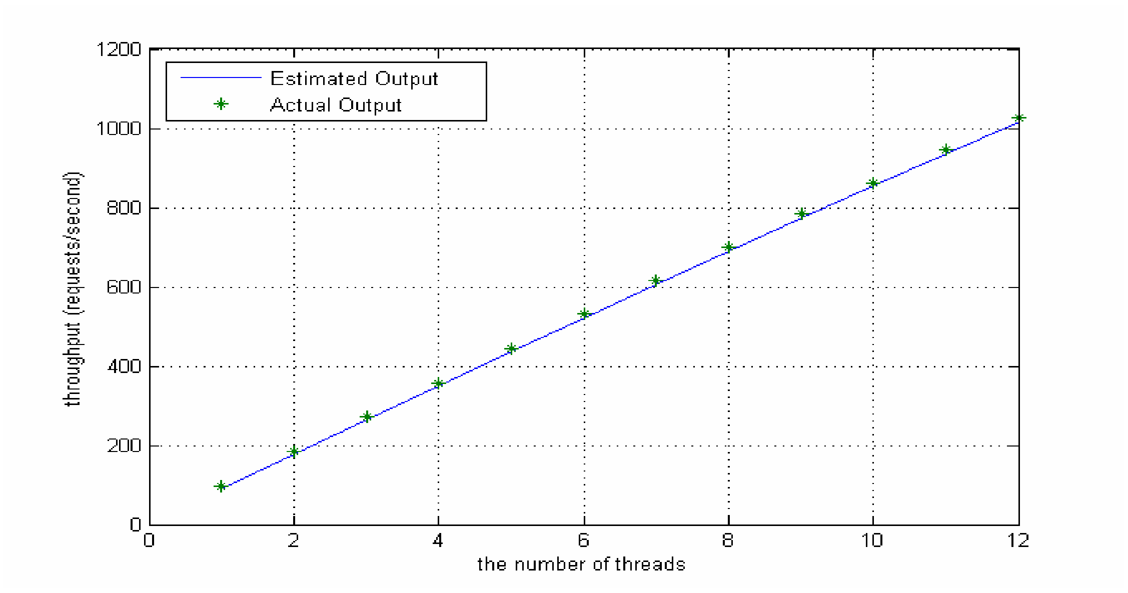


Figure 4-7 Comparison of the estimated output and the actual output

Despite the small difference between the actual and the estimated output from the above system model, the comparison demonstrates that modeling the system in this way is valid and the model is reliable to be used in the automatic control system designs. Note that the above system identification algorithm can be used in off-line or on-line modeling.

4.3.4 Feedback Control System

By making use of the black box system identification technique to model the stage thread pool, an ARX model-based linear SISO system is developed for control. This model is a linearized model, which hides uncertainties in the actual nonlinear model. The estimated model obviously cannot provide fully reliable information for feed-forward designs. However, the feedback control is able to make use of the residual between the reference and actual output to correct the model errors and provide quality-guaranteed control performance, though the target system may have some errors in the modeling. Therefore, feedback control is chosen as the control strategy in the current design. The system control model is developed as in Figure 4-8.

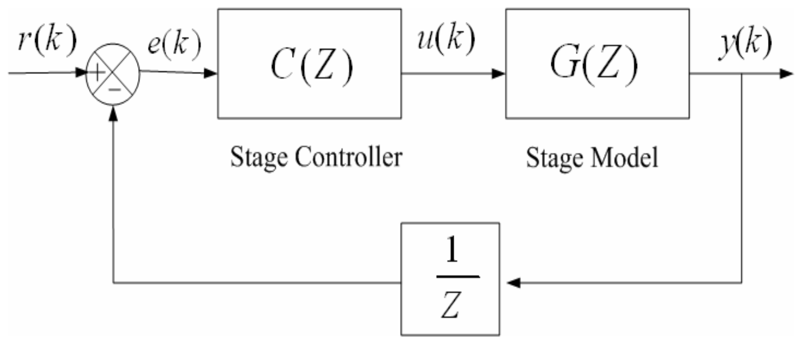


Figure 4-8 Stage Feedback Control Model

In Figure 4-8 G(z) and C(z) respectively represent the stage model and the controller model in z-plane [45], and $r(k)$, $e(k)$ and $u(k)$ denote reference, error and controlled signals at sampled time k respectively. In this design, in physical, $u(k)$ and $y(k)$ mean the change in the number of threads and the throughput respectively. When $u(k) > 0$, it means the thread pool controller will add $u(k)$ threads in the stage; otherwise, it means $u(k)$ threads are halted. Whenever the stage receives the control input, the performance is immediately changed.

Unlike the traditional feedback control system model (shown in Figure 4-4), a unit delay is placed in the feedback path in the current stage control (as Figure 4-8 shows). The delay here retains the current output for generating the residual e used in the next sampled time, and solves the algebraic loop problem. The current controlled model (Figure 4-8) makes use of the present system input to directly control the present output (illustrated in Figure 4-6), instead of using the past input and output states to estimate the current output. In terms of the control system model, the system transfer function can be developed as:

$$f_{sys}(z) = \frac{C(z)G(z)}{1 + C(z)G(z)z^{-1}} \tag{20}$$

And converting the controlled system model from the time domain to the z-plane, we have:

$$G(z) = \frac{BZ}{Z - A} \tag{21}$$

Then, the remaining work is to design the controller $C(z)$.

4.3.4.1 PID Controller Design

The original SEDA design exploits the heuristic approach to control the performance of each stage. The advantages and pitfalls of the heuristic control are discussed at length in the previous sections. Since a software system generally needs a fast reaction and stable performance, the PID control algorithm is chosen in the current control system design. Experiment results demonstrate that the PID control is efficient in practical applications. In particular, the proportional control based pre-compensator design provides an effective way to control the performance with the quality guaranteed, even when the system is running under an unpredictable dynamic loading environment.

PID is an abbreviation of proportion, integral and derivative control. PID control may be one of the simplest and the most commonly used control approaches in either academic research or industrial engineering areas. A PID model is shown in Figure 4-9.

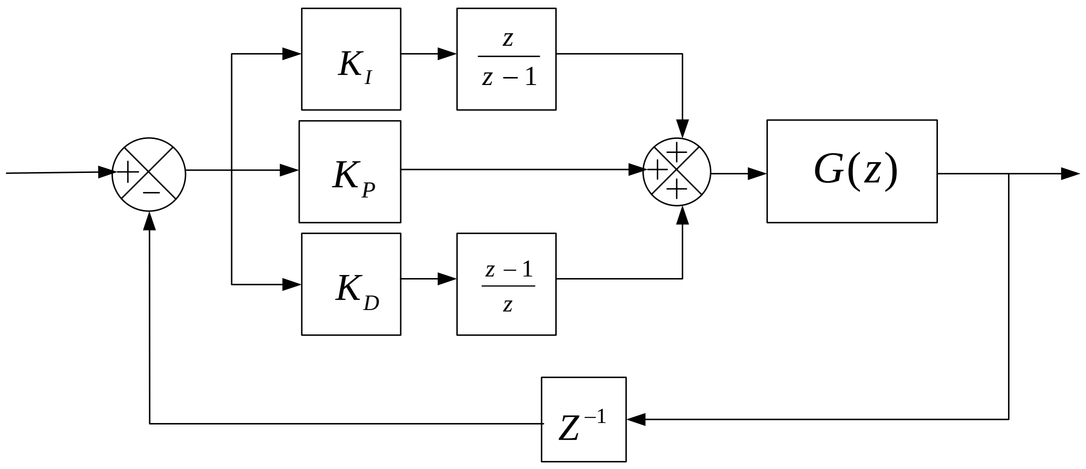


Figure 4-9 PID control on the stage

In Figure 4-9, K_I, K_D and K_P respectively denote integral, derivative and proportional control parameters. Proportional control is typically a simple amplifier with a constant gain K_P, which offers a controlled signal proportional to the actuating signal. If the gain

K_P in the controller is a variable that can be adjusted automatically following the change of the dynamic system behaviors, this control technique is the well-known gain scheduling approach, one of the advanced control techniques. In fact, gain scheduling is not only limited to real-time change on the amplifier constant K_P. It is an approach which adjusts the parameters of the controller at runtime by monitoring the operating conditions of the process [9]. However proportional control has a limitation in that it cannot remove the steady-state error, for if the control error $e(k)$ is zero, the control input $u(k)$ is also zero. In general, there are two approaches that deal with the steady-state error. One works together with integral control, and the other applies the pre-compensator. The mathematical depiction of proportional control is shown as Equation (22).

$$u(k) = K_P e(k) \tag{22}$$

Unlike proportional control, the control input in integral control is allowed to be nonzero even when the current error is zero. This allows the system to have zero steady-state error in the presence of a step change in reference inputs. In general, proportional and integral control approaches are used together, thereby using integral control to eliminate the steady-state error, as well as taking the advantage of proportional control to achieve the fast convergent speed.

In general, the integral control has the form as:

$$u(k) = u(k-1) + K_I e(k) \tag{23}$$

where $u(k)$ is the output of the integral controller and $e(k)$ is the control error. The control parameter K_I defines the ratio of the control change to the control error. Unlike proportional and integral control, which reacts based on the current error or past errors, derivative control reacts immediately to a large change in the control error.

The derivative control law has the form:

$$u(k) = K_D[e(k) - e(k-1)] \tag{24}$$

where the derivative control gain K_D defines the ratio of the input magnitude to the change in the error.

A derivative controller adjusts the control input based on the speed of error variation, which enables an adjustment prior to the appearance of even larger errors. In general, derivative control can enhance the system response time, however it usually will also result in undesirable overreactions. Since a derivative controller cannot react to a constant error, derivative control is always used in conjunction with proportional control and sometimes also with integral control [9, 30].

4.3.4.2 PID Controller in SEDA

Based on the above PID control algorithms, proportional control (P control), proportional-integral control (PI control) and proportional- derivative control (PD control) are respectively deployed in the current design.

4.3.4.2.1 Proportional Controller

We first apply the proportional control algorithm in the design. According to proportional control, the controller in P-control is a proportional constant between control error and output. Here we use K_P to represent the gain of the amplifier. Based on the system model (Figure 4-8, Equation (20) and Equation (21)), under proportion control, the system transfer function is depicted by Equation (25).

$$f_{sys}(z) = \frac{K_p \dfrac{BZ}{Z-A}}{1+K_p \dfrac{BZ}{Z-A} Z^{-1}} = \frac{K_p B}{1-(A-K_p B)Z^{-1}} \tag{25}$$

Converting the transfer function from z-plane to time domain by inversed z-transformation [45], we achieve Equation (26).

$$X(n) = K_P B(A-K_P B)^n u[n] \tag{26}$$

where $X(n)$ represents the variance of the output at time n, and the $u(n)$ represents the input signal to the system at this sample time. In this design, $u(n)$ specifically denotes the sampled reference input $r(k)$, which is the input to the control system.

In order to stabilize the system, all poles of the transfer functions are needed to be placed in the unit circle. Therefore, a constraint on the choice of the parameter values in Equation (20) is as shown in Equation (27).

$$\left|A - K_p B\right| < 1 \tag{27}$$

According to Equation (26) and Equation (27), when running time is unlimited, the system output can then be calculated by the Equation (28) as shown below

$$\sum_{n=0}^{\infty} X(n) = \sum_{n=0}^{\infty} K_p B(A - K_p B)^n = \frac{K_p B}{1 - A + K_p B} \tag{28}$$

If the system is stable, the system final output is then convergent to a final value. We say that it is the steady output of the system.

If the steady output is the expected output, that is the system can perform as expected. In mathematics, it can be represented as Equation (29)

$$R = \sum_{n=0}^{\infty} X(n) = \sum_{n=0}^{\infty} K_p B(A - K_p B)^n = \frac{K_p B}{1 - A + K_p B} \tag{29}$$

where R is the reference input or the desired target output. That is the purpose here is to seek the best value of K_P, so that the control output is the desired target. So, we let

$$1 - A + K_p B = 1 \tag{30}$$

Then we can get

$$K_p = \frac{A}{B} \tag{31}$$

When the system's steady output meets the need of the desired target, putting the above results in the system model, we have

$$y(k) = \alpha A r(k) \tag{32}$$

where α is the gain of the pre-compensator. Obviously,

$$\alpha = \frac{1}{A} \tag{33}$$

And the system model is as Figure 4-10 shows.

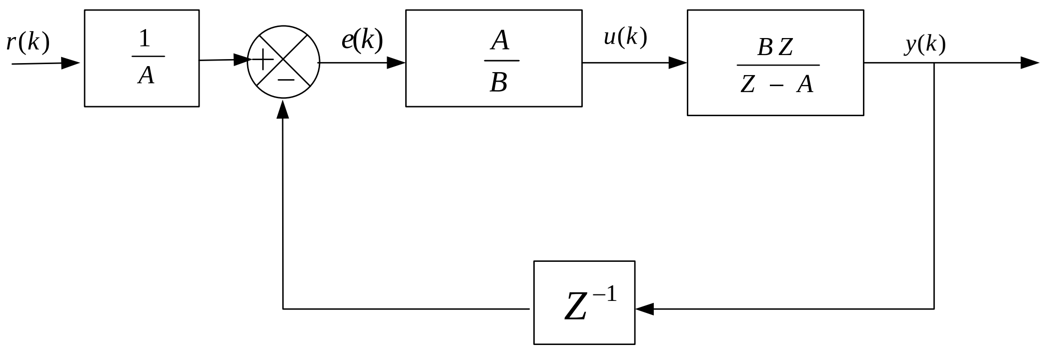


Figure 4-10 P-Control Based Pre-Compensator Control System

Based on this algorithm, we argue that this approach can eliminate the steady-state error. Figure 4-11 is the simulation result of the design

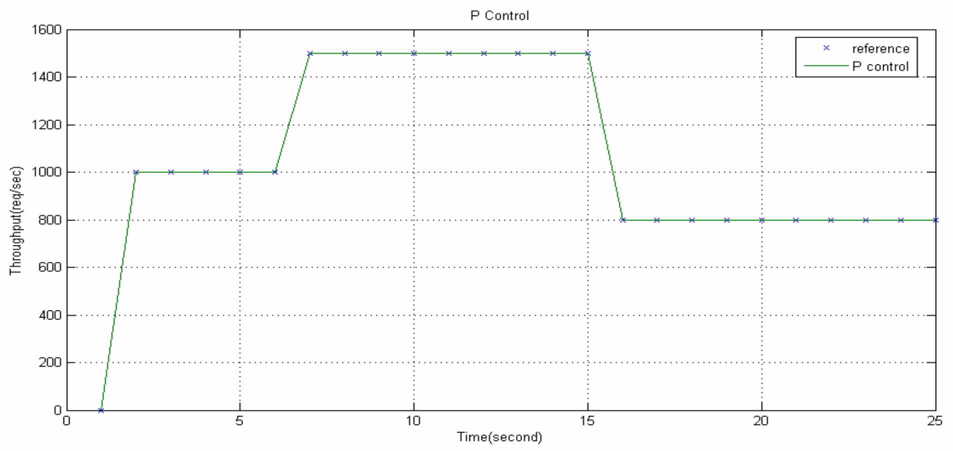


Figure 4-11 The simulation result of P control on Stage

The pre-compensator control approaches generally are not able to be used in noisy environments [30]. When the target system is modeled with high accuracy and run in a non noisy environment, the pre-compensator design can effectively control the system performance. In contrast, whenever the system has unknown disturbances or there are large errors lying between the estimated and the actual model, the pre-compensator may not work properly. In the current design, this problem is solved by the automatic modeling mechanism. The auto-model mechanism real-time monitors and models the performance of the controlled system. Once the working environment or the model of the target system is changed, the auto-model mechanism can then make use of the new information of the controlled system to update the controller parameters immediately, so that it is able to guarantee the accuracy of the controller parameter configurations and the control performance.

4.3.4.2.2 Proportional- Integral design

The above section briefly introduced PI control. There have been a large number of discussions on how to get the best values of these controlled parameters, meeting the need of the expected output. However, most of the previous achievements are limited to a continuous system or the discrete systems that are not affected too much by zeros. For these reasons, we here propose a new approach to seek the optimal values of the control parameters for more general cases.

From a high-level overview, the design process is concluded as the following diagram shows (Figure 4-12)

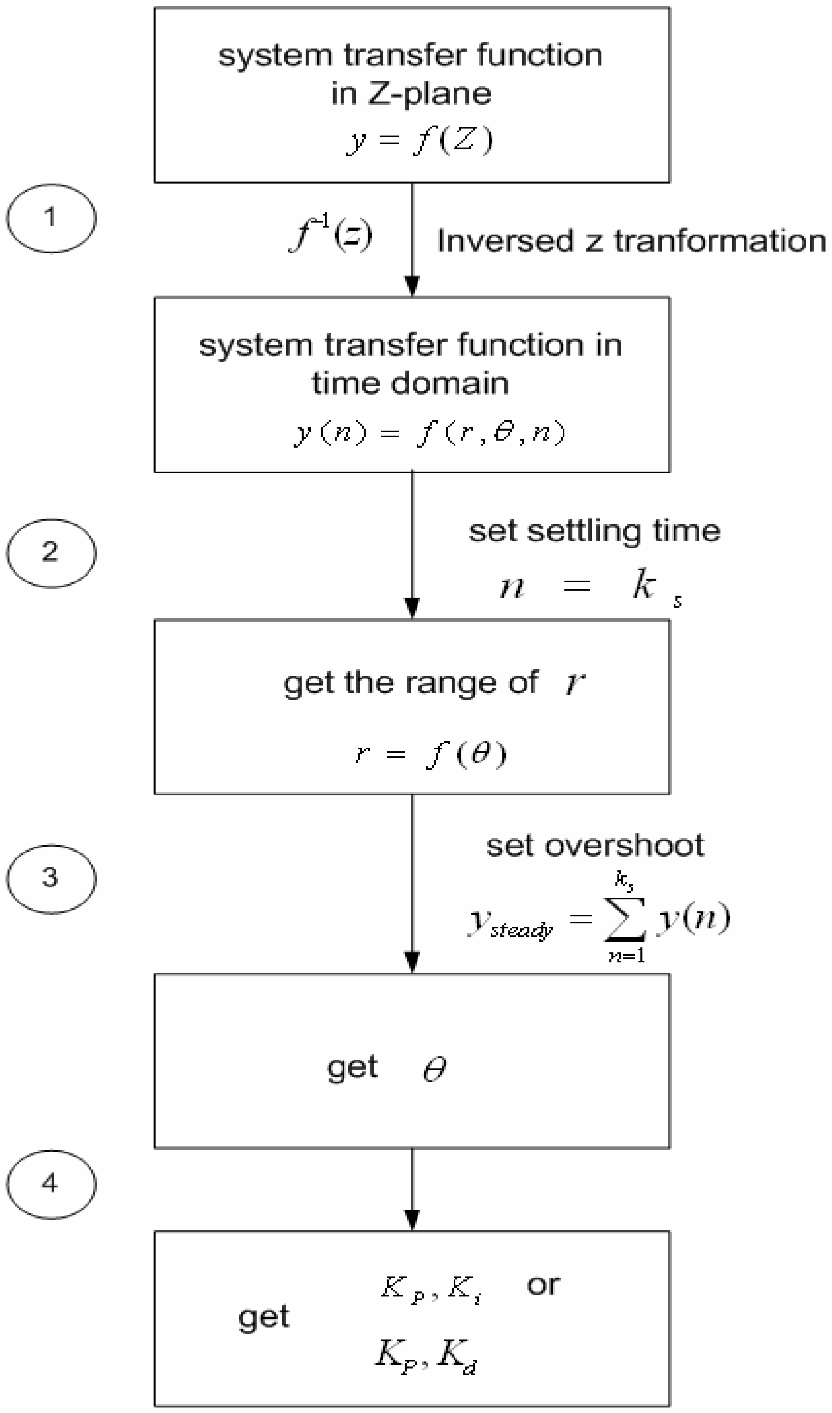


Figure 4-12 The process that achieves the optimal values to configure controller

In Figure 4-12, r and θ respectively denote the distance and the angle of the location of the poles in polar coordinates [45]. The definitions of these parameters are detailed below. In the design, we firstly model the system in the z-plane, and then make use of the inversed z-transformation to transform the system model into the equations represented by time. Finally, putting the desired performance targets of setting time and overshoot into the system model, the potential appropriate choices of K_p, K_i and K_p, K_d are achieved.

When using PI control algorithm, the system transfer function is

$$f_{PI}(z) = \frac{B(K_p + K_i)Z^2 - BK_p Z}{Z^2 - (A+1-K_p B - K_i B)Z + A - K_p B} \quad \textbf{(34)}$$

Converting to the equation in the time domain, we have

$$X_{PI}(n) = \left[\frac{-B(K_p + K_i)p_1 + BK_p}{p_2 - p_1}(p_1)^n + \frac{-B(K_p + K_i)p_2 + BK_p}{p_1 - p_2}(p_2)^n \right] u(n) \quad \textbf{(35)}$$

Where P_1 and P_2 respectively denote the poles of Equation (34), and $X(n)$ is the change of the output resulting from the input $u(n)$.

From Equation (35), we get Equations (36 to 38).

$$\nabla = (A+1-K_p B - K_i B)^2 + 4(A - K_p B) \quad \textbf{(36)}$$

$$P_1 = \frac{-(A+1-K_p B - K_i B) + \sqrt{\nabla}}{2} \quad \textbf{(37)}$$

$$P_2 = \frac{-(A+1-K_p B - K_i B) - \sqrt{\nabla}}{2} \quad \textbf{(38)}$$

In terms of the stability requirement, we need $\nabla < 0$. If we represent P_1 and in polar coordinates, P_1 and P_2 also can be represented as Equation (39) as shown in Figure 4-13.

$$P_1 = |r| e^{j\theta} \text{ and } P_2 = |r| e^{-j\theta} \quad \textbf{(39)}$$

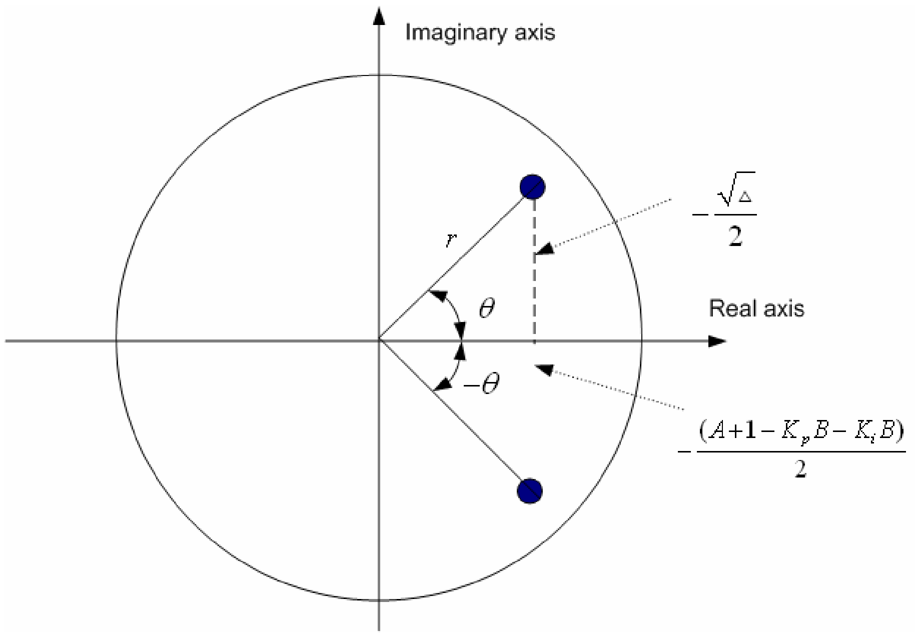


Figure 4-13 Polar Coordinator

The transfer function of the system then can be written as:

$$X_{PI}(n)=\left[\frac{BK_p-B(K_p+K_i)|r|e^{j\theta}}{|r|e^{-j\theta}-|r|e^{j\theta}}(|r|e^{j\theta})^n+\frac{BK_p-B(K_p+K_i)|r|e^{-j\theta}}{|r|e^{j\theta}-|r|e^{-j\theta}}(|r|e^{-j\theta})^n\right]u(n) \quad \textbf{(40)}$$

And because of

$$e^{j\theta n}=\cos(\theta n)+j\sin(\theta n) \quad \textbf{(41)}$$

$$X_{PI}(n)=\left[\frac{B(K_p+K_i)|r|^{n+1}\sin(\theta n+\theta)-BK_p|r|^n\sin(\theta n)}{|r|\sin\theta}\right]u(n) \quad \textbf{(42)}$$

We set the desired settling time and overshoot for the system.

When it is at the settling time n, $X_{PI}(n)$ = 0, that is

$$B(K_p+K_i)|r|^{n+1}\sin(\theta n+\theta)-BK_p|r|^n\sin(\theta n)=0 \quad \textbf{(43)}$$

$$B(K_p+K_i)|r|^{k_s+1}\sin(\theta k_s+\theta)-BK_p|r|^{k_s}\sin(\theta k_s)=0 \quad \textbf{(44)}$$

$$\frac{BK_p}{B(K_p+K_i)}=\frac{\sin(\theta k_s+\theta)}{\sin(\theta k_s)}|r| \quad \textbf{(45)}$$

From Figure 3.15, we have

$$K_p = \frac{A - r^2}{B} \tag{46}$$

$$K_i = \frac{r^2 + 1 - 2r\cos\theta}{B} \tag{47}$$

$$[1 - 2\frac{\sin(\theta K_s + \theta)}{\sin(\theta K_s)}\cos\theta]r^2 + \frac{\sin(\theta K_s + \theta)}{\sin(\theta K_s)}(1 + A)r - A = 0 \tag{48}$$

We achieve the range of r by solving Equation (48).

And we know that the steady output of the system occurs in k_s. Therefore, the system's final steady output equals

$$\sum_{n=1}^{k_s} y(n) \tag{49}$$

For stability, the $0 < r < 1$ and $\theta \in [0, \frac{\pi}{2}]$

We will have multiple sets of r and θ that meet the need of the performance demands. Note that each r has a corresponding θ, using it as a pair.

Selecting one of these and putting it back into Equation (46) and Equation (47), we can calculate the values of K_p and K_i that are the desired configuration to produce the desired performance.

Using a stage model to run the simulation in Matlab, we obtain the result shown in Figure 4-14.

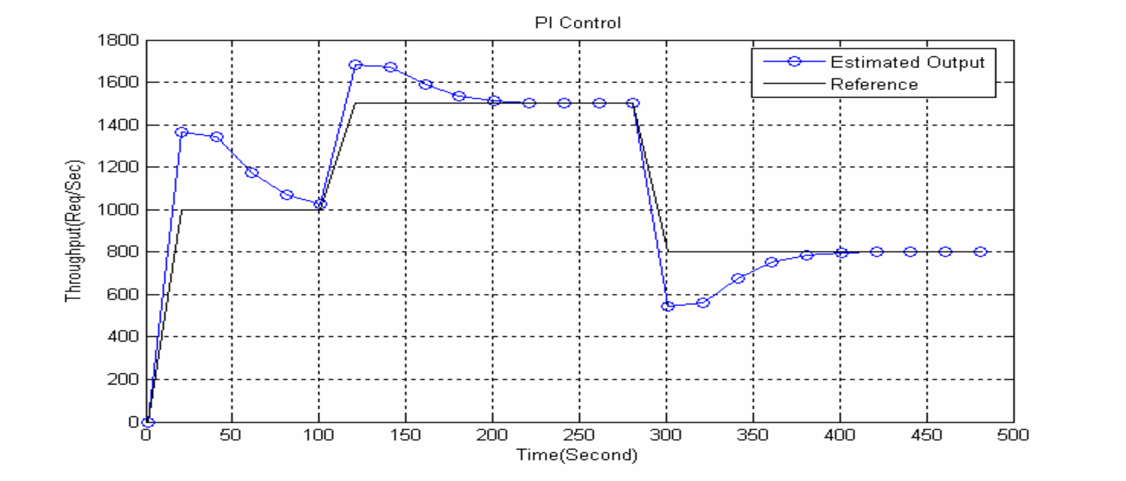


Figure 4-14 The simulation result of PI control on the stage

The overshoot here obviously cannot meet the performance demands, but it is the best overshoot that we can achieve, for the requirements of the stability limit for the available choices of r and θ.

4.3.4.2.3 Proportional-Derivative

The way to choose correct values for the PD controller parameters is very similar to the above PI control strategy.

In PD control, the transfer function is shown as Equation (50)

$$f_{PD}(z)=\frac{B(K_p+K_d)Z^2-BK_dZ}{Z^2-(A-K_pB-K_dB)Z-K_dB} \tag{50}$$

And the time domain behavior is then described as Equation (51)

$$X_{PD}(n)=\frac{B(K_p+K_d)r^{n+1}\sin(\theta n+\theta)-BK_dr^n\sin(\theta n)}{r\sin\theta}u[n] \tag{51}$$

where θ and r are the same as the above PI control algorithm designs, denoting the angle and the distance of the pole located in the z-plane.

Following the computation flowchart (Figure 3.12), we obtain

$$r = \frac{A\sin(\theta K_s + \theta)}{2\cos\theta\sin(\theta K_s + \theta) - \sin(\theta K)} \tag{52}$$

Finally, we obtain the optimal values of the configuration, by applying Equation (53) and Equation (54).

$$K_D = -\frac{r^2}{B} \tag{53}$$

$$K_p = \frac{A - 2r\cos\theta}{B} - K_D \tag{54}$$

And the simulation result is shown in Figure 4-15

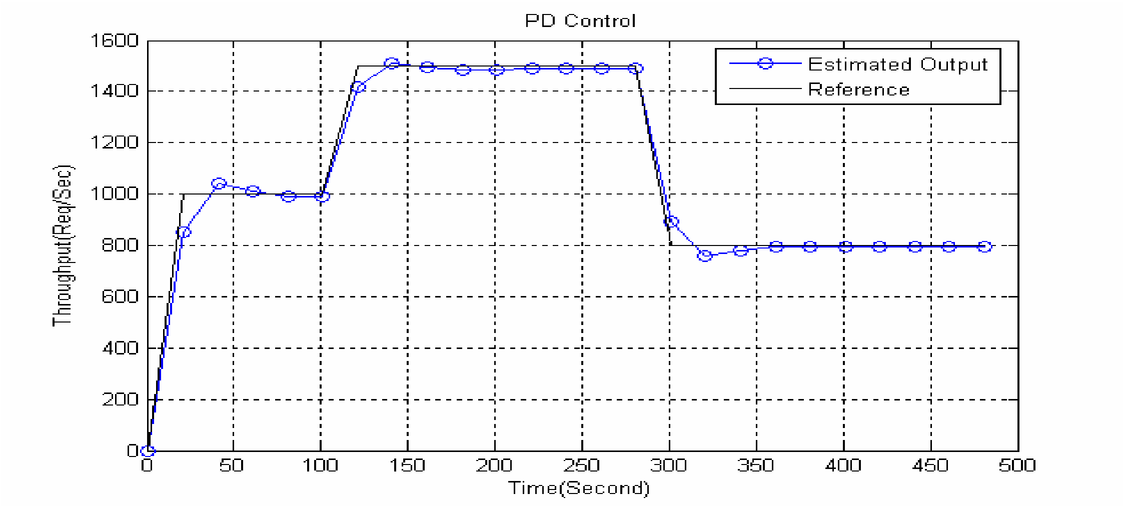


Figure 4-15 The simulation result of PD control on the stage

The simulation results are the same as the above theoretical proof, demonstrating that our theorem design is correct. In the next chapter, we will put the above design into a SEDA-based web server, benchmarked by SPEC99 [51], and use the results obtained from real-life to compare against our theoretical arguments.

Chapter 5 Tests, Evaluation and Analysis

In this chapter, we show the experiment results of putting the above designs into practice using a SEDA-based web server. The experiment involves two parts. The first validates the design and evaluates the control approaches on throughput. The other part demonstrates the application of the control framework to provide fair service to unpredictable dynamic loadings. This experiment includes an example of control over the 90th percentile response time in order to meet the changed dynamic target at runtime, and the control over the robust performance when a large number of arbitrary loading is added onto the server.

Our design is firstly validated by implementing the above control strategies into a SEDA-based web server (Figure 5-1) and evaluated via benchmarking. In these experiments, the test bed 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 as the JAVA platform running Linux kernel v2.6. The client is running a synthetic workload generator based on the SPECweb 99 [51] testing suite.

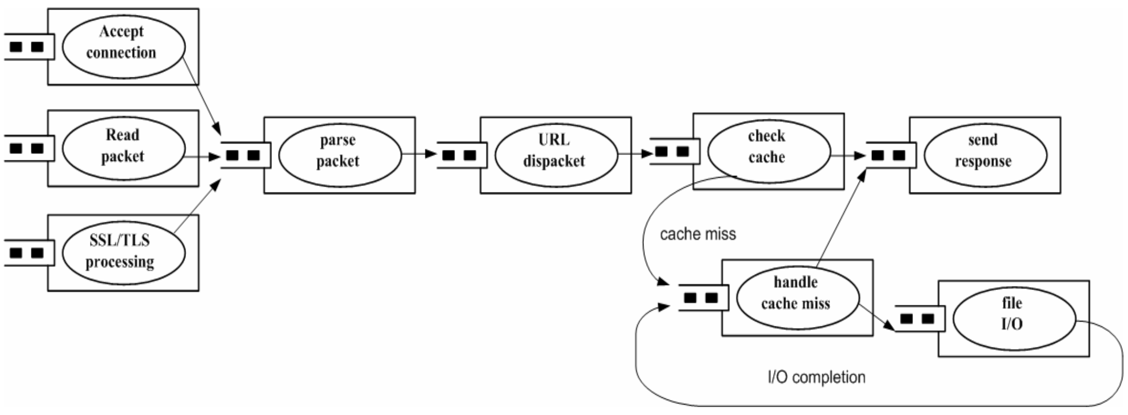


Figure 5-1 SEDA-based web server

5.1Throughput Control and Evaluations

This experiment evaluates the P, PI and PD control over the throughput for the SEDA-based Web Server. Results are illustrated in Figure 5-2. In these experiments, 500 clients are used to concurrently assess the SEDA web server, and the web server is required to automatically adjust the number of threads in the stage thread pool to ensure the demanded throughput at runtime in terms of the control mechanism introduced in 4.3.2 and 4.3.4.

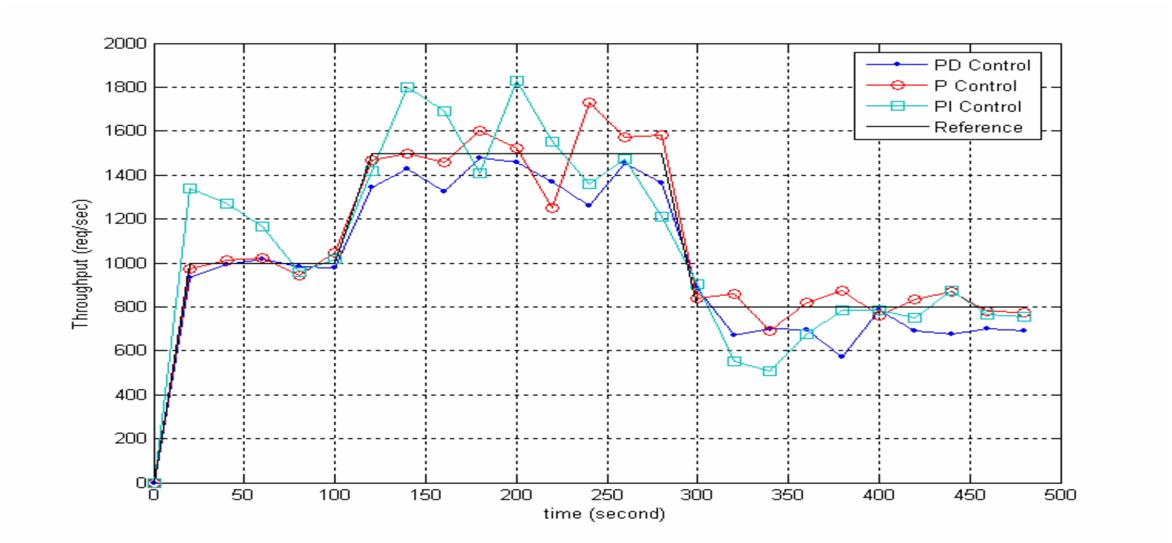


Figure 5-2 The performance of P, PI and PD control

In Figure 5-2, the horizontal axis represents the time, and the vertical axis stands for the throughput. The performance targets in the three experiments are the same, represented by the solid black line. Theoretically, as demonstrated in the last chapter, when the modeling is correct and the controller parameters are configured with optimal values, P, PI and PD can all meet the performance demands. Here, we first use P, PI and PD's actual control results to compare against its own simulation result, as the figures below show.

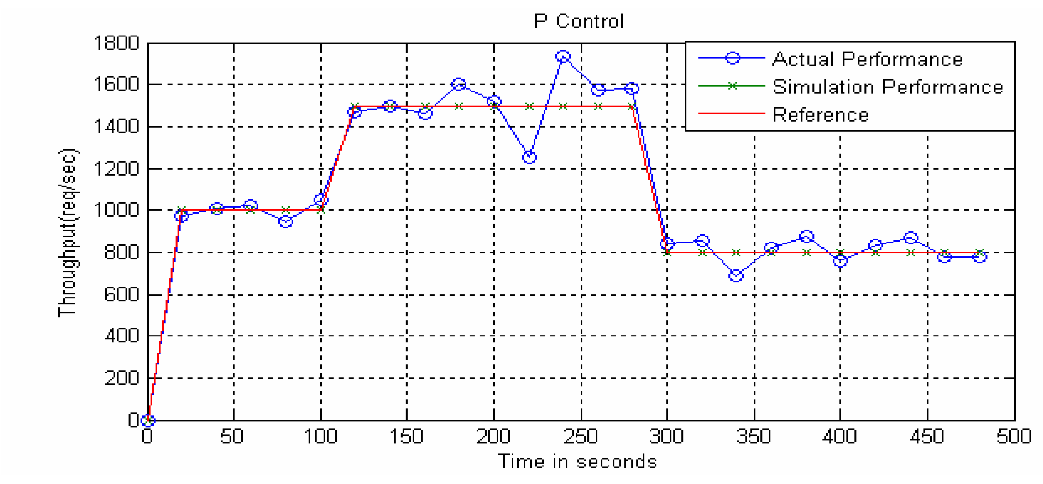


Figure 5-3 the performance of P control in simulation and practice

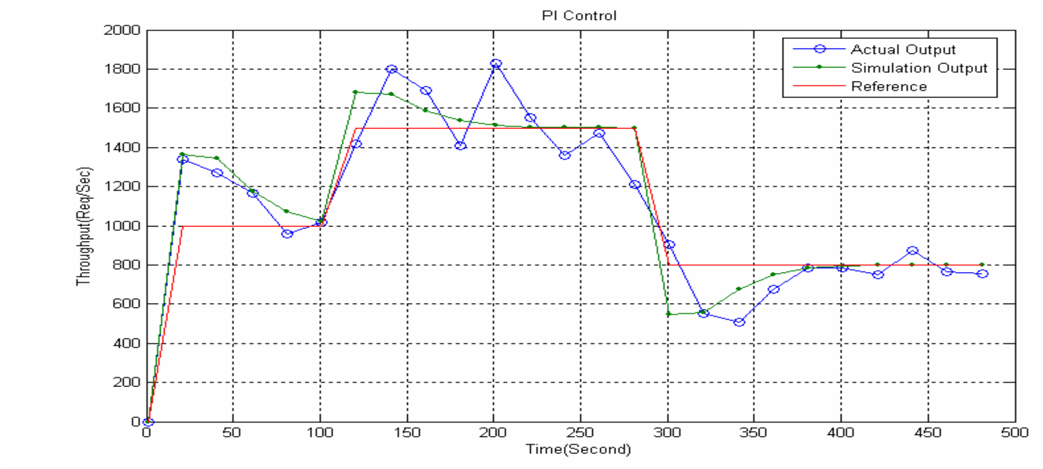


Figure 5-4 the performance of PI control in simulation and practice

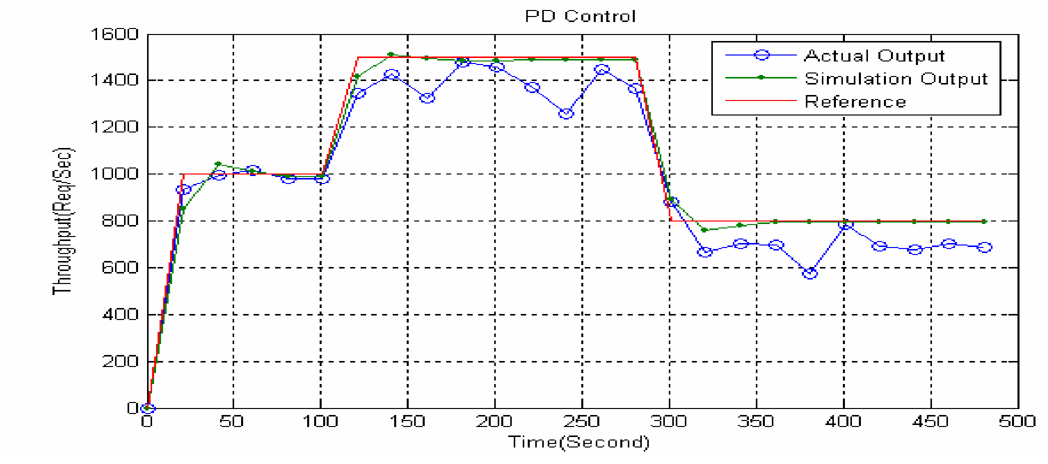


Figure 5-5 the performance of PD control in simulation and practice

As the above comparisons demonstrate, the actual performance of our design is almost the same as the simulation results. Despite some oscillations in behavior, the controllers are effective at keeping the performance near the target. Note that the oscillation here may be a result of Java’s garbage collection [54].

From this comparison, it can be seen that the performance of the P and PD controller are better than PI, and the design of the P-control-based pre-compensator is best. P-Control also provides the best performance in both theoretical simulation and practical applications. In spite of the outstanding performance of the pre-compensator design, however, the pre-compensator control model is usually not well suited for the system running under the dynamic loading environment. Here, the problem is solved by making use of the automatic modeling mechanism, in which the controller parameters are adjusted in real-time following the change in the target system, so that the control system maintains its accuracy, matching the present model of the control target.

We then choose the P control to compare against the original heuristic control used in original SEDA. The comparison result is shown in Figure 5-6.

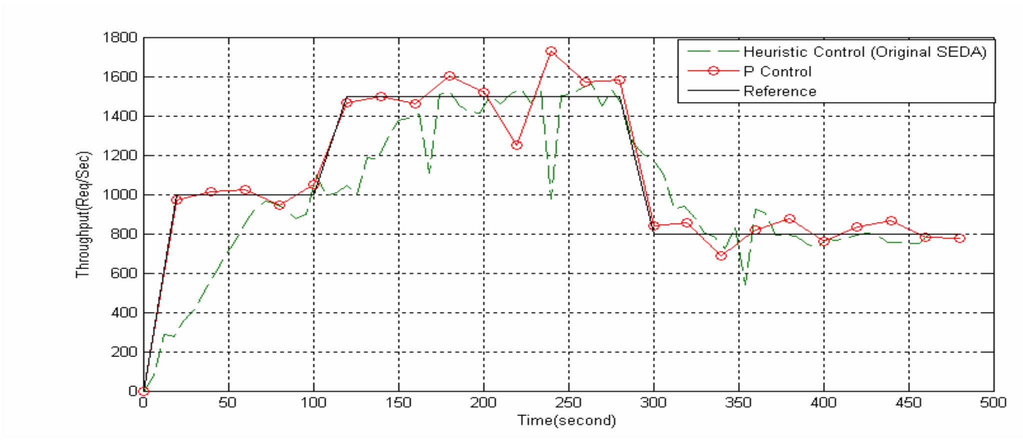


Figure 5-6 Comparison of the P-Control and Heuristic Control

The performance of the heuristic control depends on the fixed control parameter settings. When the working environment or the performance target is changed at runtime, the heuristic control cannot adaptively adjust its configurations to meet the dynamic goal. Compared with this, the P-control design developed here not only shows a very fast convergent speed to catch the new dynamic target (as Figure 5-6 shows), but also guarantees the quality of the performance with the real-time optimal parameter settings. It also avoids the time-consuming and error-prone manual configurations.

This experiment shows that P-control-based pre-compensator design performs well for the stage control. In fact, a thread pool model, like Apache, has the same model as a stage. It implies that this design also has a strong potential to support other thread-based systems if required.

5.2 Controlling the 90th Percentile Response Time

Based on the best control approach, P-Control, we firstly test the fair service of the system when the number of requests is not overloaded. The tests are built up on a partly-open loop benchmark [10], and include two cases. The first one controls the 90th percentile response time to meet the target changed at runtime. The second one keeps the response time constantly at the target under an unpredictable dynamic loading environment. The controller in each stage executes the adjustment every 2 seconds.

Most of the benchmarks currently in use are grouped into two catalogues. One is the “open loop” benchmarks such as httperf [32]. 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. Another type of the benchmark is the “closed loop” model. “Clients” in this kind of benchmark will only send new requests until the response of the last request has come back, like TPC-C [53] etc. Benchmarks of this type can provide constant workload on the server side. A pitfall of this kind of benchmark is that it cannot put increasing loadings onto the server. If it is needed to emulate a dynamic loading environment, a couple of such benchmarks are required to run alternatively during the experiment, which may consume a great deal of resources. In terms of some failures of the “open loop benchmark” and the “closed loop benchmark”, a “partly open loop” benchmark is proposed in [10]. The “partly open loop” system here is developed based on the prototype of SPEC 99. Whenever the response of the most recent request is not received within the expected time, the client will send a new request to the server. The "partly open loop" benchmark simulates a dynamic environment and keeps putting heavier loads onto the server with less resource consumption. The "partly open loop" model has the advantage that it can not only perform as a "closed loop" benchmark that places a fixed and finite number of requests on the server, but also can behave as other "open loop" benchmarks to simulate an unpredictable dynamic loading environment.

5.2.1 Follow 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 emulates 250 clients that 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 experiment result is shown in Figure 5-7. We here use "reference", a term from control theory, to represent the "target performance". It can be seen from the results that, during the experiment, the server is able to effectively adjust the resources to produce the desired service performance following the change of the desired target, and maintain the percentage response time almost stay around the target with a fast convergent process.

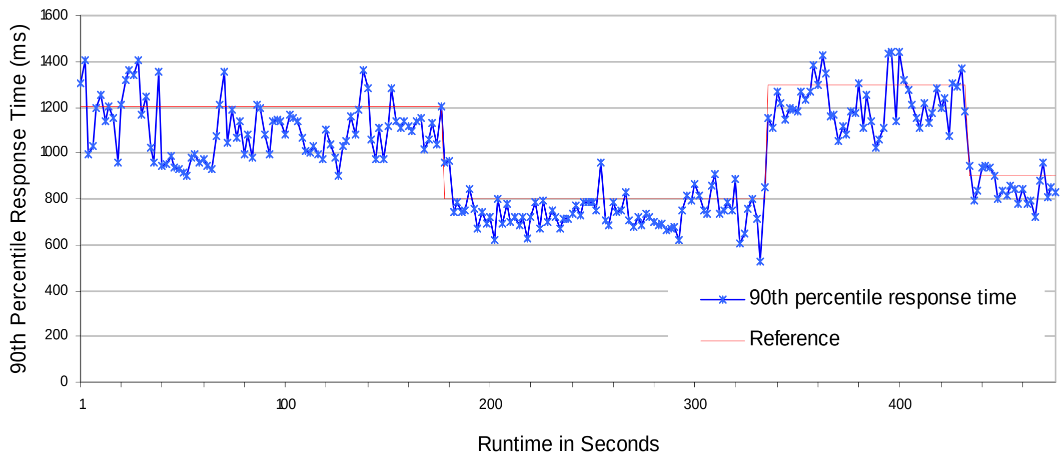


Figure 5-7 Adjust the 90th Response Time On-The-Fly

5.2.2 Maintain stable 90th percentile response Time

This experiment tests the performance of our design in keeping a constant percentile response time during unpredictable dynamic loading environments. In terms of the partly-open loop workload generator, the number of requests which stay in the web server will be fixed and finite as the received response time is less than the target benchmark; however, once the response time is less than the expected target, the number of requests will keep increasing. In order to provide the desired response time to the increasing request loads, our automatic control mechanism will execute resource management to increase the utilization of the system resource so as to reduce the queuing time of requests, thus ensuring the service performance meet the requirement. Experiment results demonstrate that, as the convergent speed is faster than the increase in requests, our design is effectively able to ensure the fair service to the incoming requests.

In the experiment, four groups of clients, of size 100, 150, 100, and 50 respectively, are randomly added onto the server at different times, and each client is independent of the others. 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 5-8.

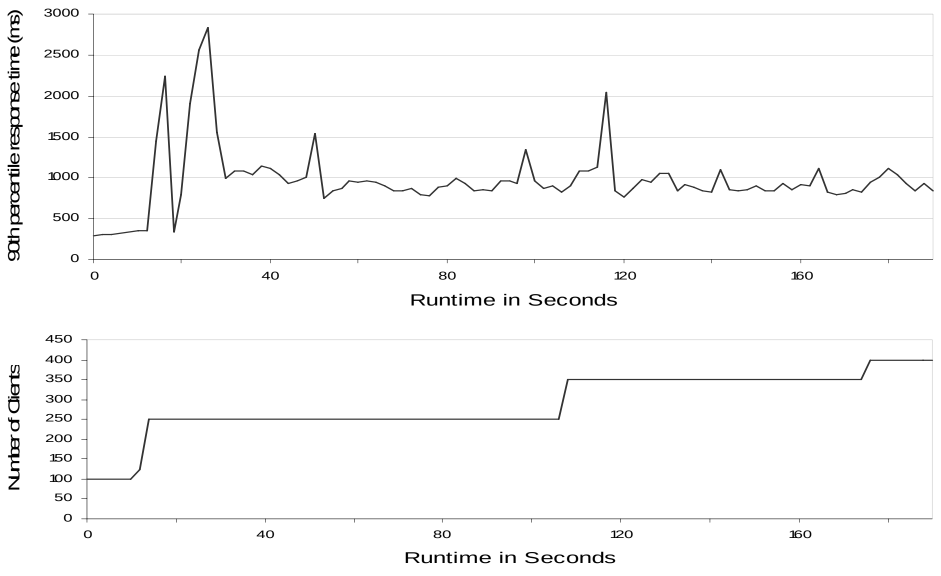


Figure 5-8 the Fair Service under Dynamic Workload

As can be seen from the results, instead of using admission control on the arrival rate at the source, management over system resources can alternatively provide the desired service performance to 90% requests, guaranteeing the fair service performance under fluctuating loads. 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 lag for a feedback control system to react to the change in a dynamic real-time 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 increasing request loads, the system will take longer time (2~3 periods in this experiment) to process these new requests, resulting in such huge spikes as which shown in the interval between the 14th and 32nd seconds. Response delay indeed is a pitfall 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.

In addition, this experiment reveals a relationship between the response time, arrival rate and the departure rate of requests. Figure 5-9 illustrates the difference of the arrival and departure rate in real-time. Referring to the response time shown in Figure 5-8, the experiment demonstrate the fact that the response time is not directly determined by the number of requests accepted by the system or the processing rate in the system, but resulted from the balance of the arrival rate and the departure rate of the requests. In one sense, this difference between the arrival and departure rates can be regarded as the total queue-fill-level in the system from the end to the front.

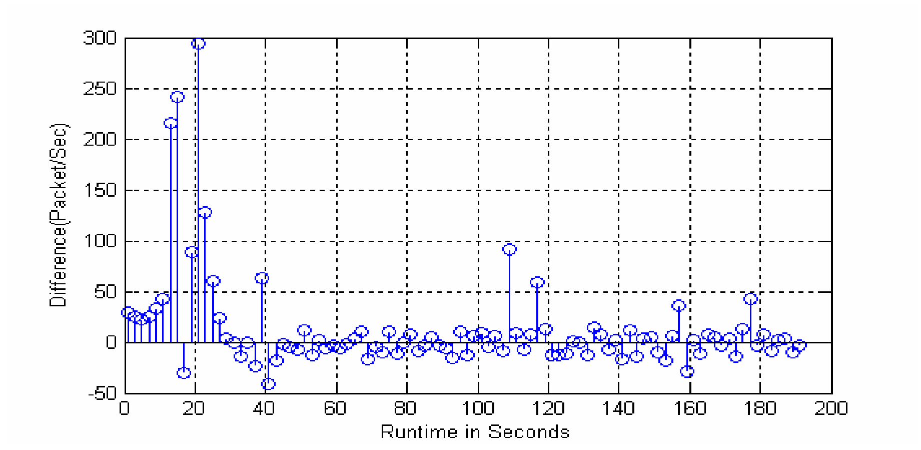


Figure 5-9 The Difference of the Arrival Rate and Departure Rate (positive value means Arrival Rate is greater than Departure Rate)

Figure 5-10 shows the distribution of the response time in this experiment. It can be seen that 90% percentile of the requests receive the response in less than 1160 ms, almost meeting our control target.

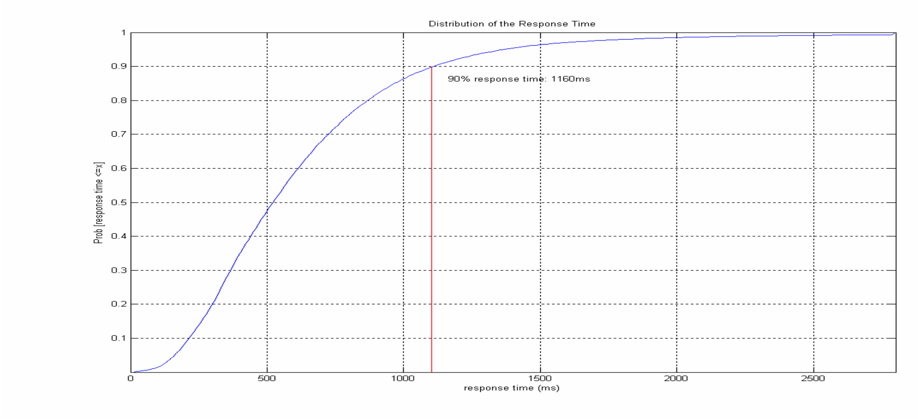


Figure 5-10 Distribution of the Response Time

Based on these experiment results, we argue that our design is an effective approach to providing explicit quantitative control on fair service for high concurrency. Even when the system runs under fluctuating loads with uncertainties, our design is still able to adaptively control the performance effectively to meet the target and show good robustness.

5.3 Compared with the Original SEDA in an Open-Loop Benchmark

In this evaluation, we will compare our design to the original SEDA-based web server that uses the admission controllers with round robins [59]. Both systems in the test are required to produce a stable percentile response time under heavy fluctuating loads.

In this testing, the original SEDA is respectively benchmarked with over-allocating and optimal-allocating threads. The setting of over-allocating threads will place more threads in the system than the actual demand of the requests, whereas the optimal-allocation is to configure each stage with an optimal thread pool size according to our long-time tests and experience-based estimation.

The sample period of our automatic controller is 30 seconds and the admission control used in original SEDA is 2 seconds. The threshold of our early selective dropper is set at 80ms and the weight w used in Equation (3) chooses the mean value of 0.5. In the experiments, the servers under test conditions are required to provide the response time in less than 100ms to 90% of the requests. On the client side, an "open loop" request generator is employed to produce the workload. A generator emulates 200 independent clients in the start, and will steeply increase to 400 clients (at the time of the 180th second) during the experiment. Each client will send a new request every 200~600ms randomly, placing unexpected fluctuating loads onto the server side. The testing time for each system is the same.

First of all, we compare the distribution of response time produced by these systems (Figure 5-11, Table 1 and Table 2). As a note, because requests are randomly dispatched by clients, the amount of requests onto each server system would be a bit different, though the testing time is the same in three experiments. Table 1 and Table 2 detail the experimental data; Figure 5-11 (a) and Figure 5-11 (b) respectively show the distribution of the response time of the processed requests and of all requests sent to the server. In Figure 5-11, the response time of the rejected requests is regarded as unlimited and the number of these requests is counted to plot the distribution.

The comparison shows the difference of the percentage of processed requests that can achieve the desired response time in these systems. It can be seen from Figure 5-11, Table 1 and Table 2 that, of either optimal-allocating or over-allocating threads, over 90% of the processed requests in the original SEDA can achieve the quality of service, whereas, just 84.9% meet the target when using the automatic control system. However, to achieve this target, the original SEDA respectively rejected 21.7% and 52.8% of the incoming requests, but in our mechanism the rejection rate is only 4%. The reason for the high rejection rate of the over-allocation can be explained by the heavy overheads associated with a large number of threading switches, especially when the number of requests increases to a certain degree. If taking the rejected requests into account, among all the client requests sent to the servers, the percentage of requests that can get the expected response time are respectively 81.2%, 74.6% and 52.8% under our automatic control mechanism, admission control with optimal-allocating and with the over-allocating threads.

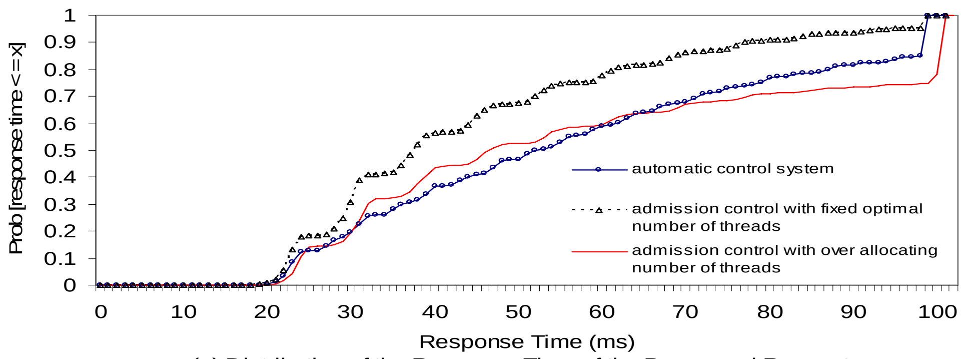
These comparisons indicate that, admission control can effectively ensure the response time to the processed requests, but may unfairly reject a large number of client requests; on the other hand, under a similar loading, our design not only can guarantee the service performance to requests, but also can provide the required service to larger amount of requests, significantly reducing the rejection rate. As a result, requests can receive a truly fairer service in terms of our design over resource management. In this test, that our automatic control system just provides 80s% requests the demanded service, not reaching the 90% requirement, is because of the setting of the control period and the test duration, which will be detailed discussed following.

Table 1 the Recorded Data in the Test

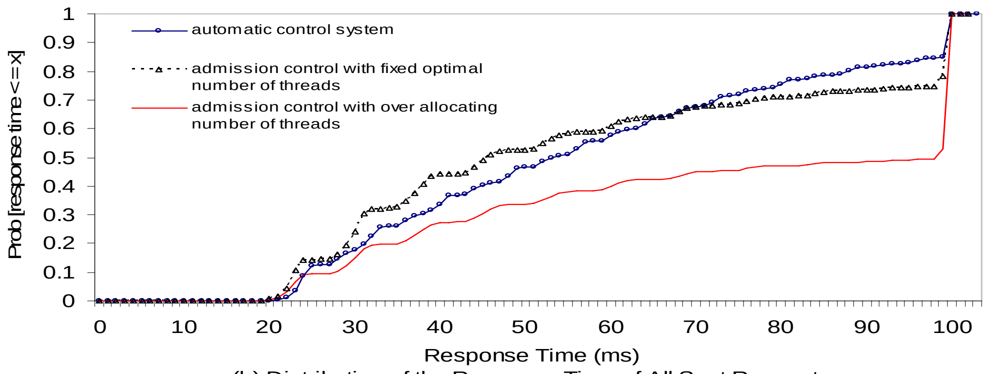
	Number of total incoming requests	Number of requests that can get the response time (<100ms)	Number of processed requests	Number of rejected requests
The Automatic Control System	462934	375952	442735	20199
Admission Control with Optimal Allocating Threads (Original SEDA)	400507	299083	313639	86868
Admission Control with Over Allocating Threads (Original SEDA)	504676	248770	266684	237992

Table 2 the Calculated Data for Comparison

	Percentage of requests receiving the satisfied service among processed requests	Percentage of requests receiving the satisfied service among all incoming requests	Rejection Rate
The Automatic Control System	84.9%	81.2%	4%
Admission Control with Optimal Allocating Threads (Original SEDA)	95.3%	74.6%	21.7%
Admission Control with Over Allocating Threads (Original SEDA)	93.2%	52.8%	52.8%



(a) Distribution of the Response Time of the Processed Requests



(b) Distribution of the Response Time of All Sent Requests

Figure 5-11 the Distribution of the Response Time

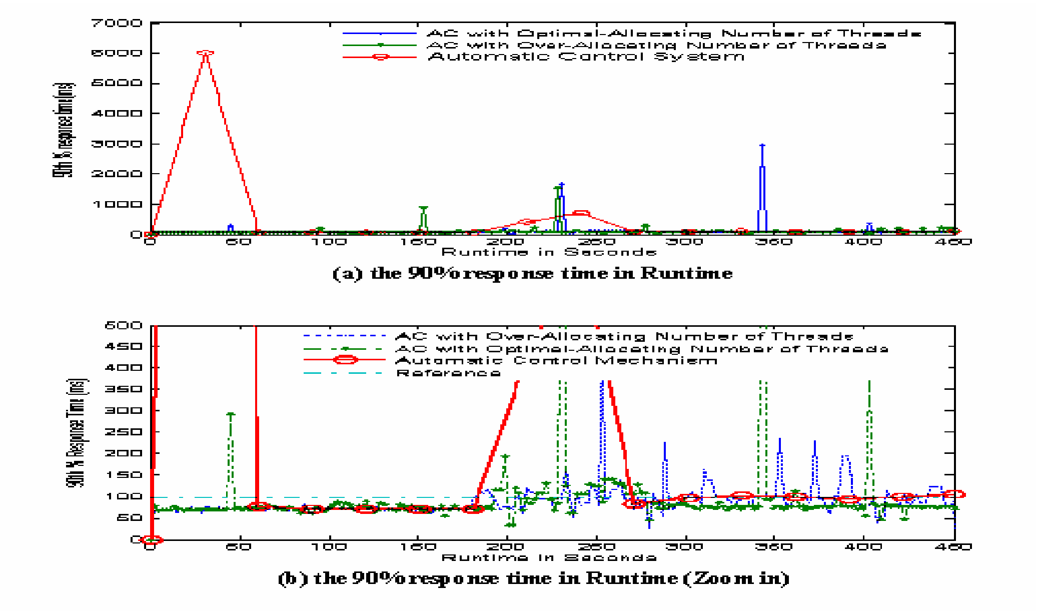
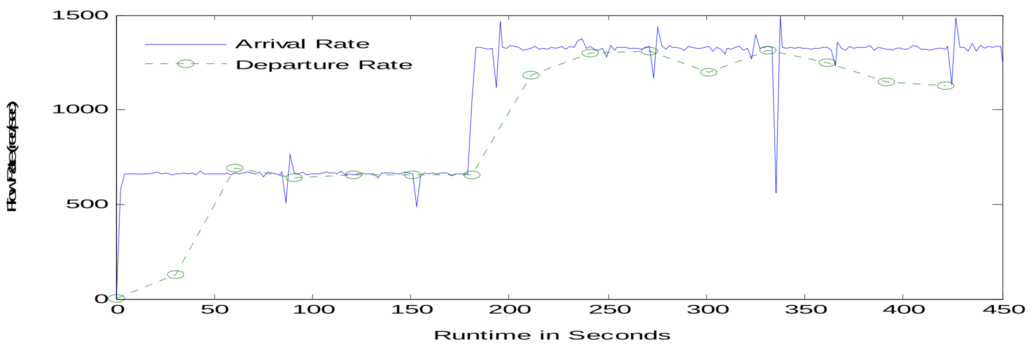


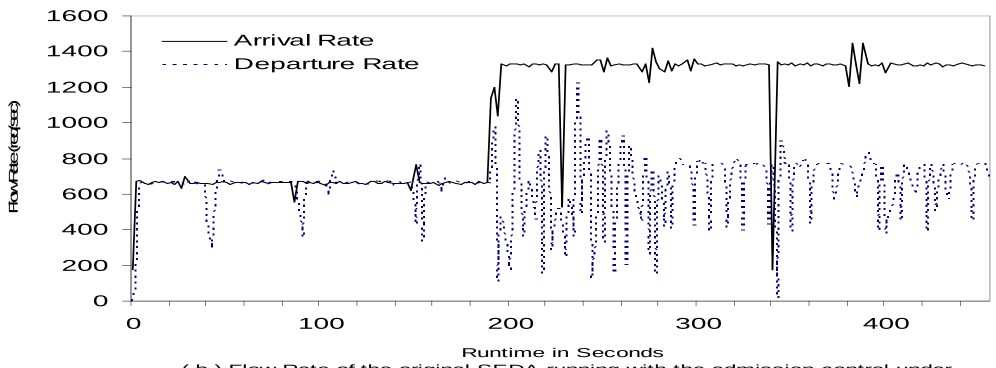
Figure 5-12 the 90the Response Time in Runtime

Figure 5-12 show the real-time change of the 90th -percentile response time in experiments. According to the results shown in Figure 5-12 (b), that tightens the shot of Figure 5-12 (a), it can be seen that our automatic control mechanism is effectively able to ensure the desired service performance for 90% requests, providing a demanded fair service with high accuracy and good stability at real-time by adaptively adjusting the system resources. In Figure 5-12 (a), that a huge spike appears during the runtime is because our control mechanism deploys feedback control strategy and chooses a long control period (sampled time) at 30 seconds. As a large number of requests are added onto a server system, the automatic controller reacts to the change at the next sample period; before the system adjusts the thread pool size to the correct values, many requests would suffer long queuing or be rejected. In this experiment, in terms of the long control period and short test duration, the number of such requests that experience bad quality of service during the huge spike degrades the final result of the percentage of requests that can receive the desired service during the whole test process, and so the data in Figure 5-11 and Table 1shows our design yet to reach the 90th percentile response time target.

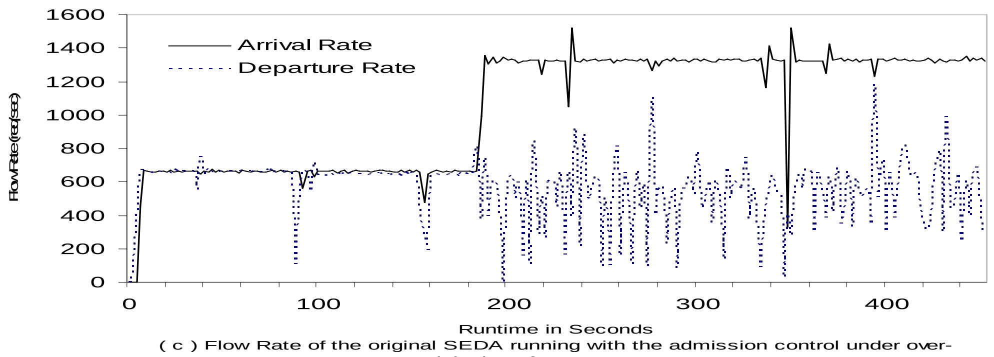
Secondly, we compare the throughput of the servers. The experiment results are shown in Figure 5-13. When the number of requests increases, our control mechanism over fairness is able to optimize the system resource in order to follow the change in the request rate, thereby adaptively supporting high concurrency. In contrast, under the same loadings, the original SEDA will only make use of the admission control mechanism to reject the incoming requests for providing the required response time to the accepted requests. As a result, although the admission control in the original SEDA can guarantee the response time of the accepted requests, it will unnecessarily reject a large number of requests.



(a) Flow Rate of SEDA under Hybrid Control
(Sample Period of the Departure Rate is 30 sec)



(b) Flow Rate of the original SEDA running with the admission control under optimal provisioning number of threads



(c) Flow Rate of the original SEDA running with the admission control under over-provisioning of system resources

Figure 5-13 the Comparison of the Flow of Rate

Thirdly, with the use of the global control framework, our auto-control system is able to balance the workload across the staged network. The rejection of requests is only used at the first stage. In contrast, under the same loading, the original SEDA will reject requests and perform at an unequal service rate in each stage, thereby resulting in a significant degradation of the overall performance (as shown in the comparison in Figure 5-14 and Figure 5-15). As a result, the resource consumed by the rejected requests in our design is much less than in the original SEDA.

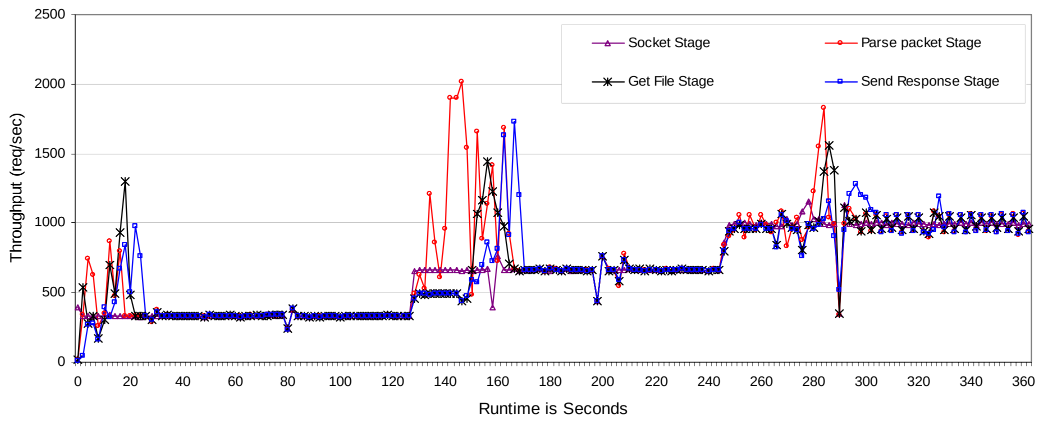


Figure 5-14 Performance of the Stages under the Automatic Fairness Control Strategy

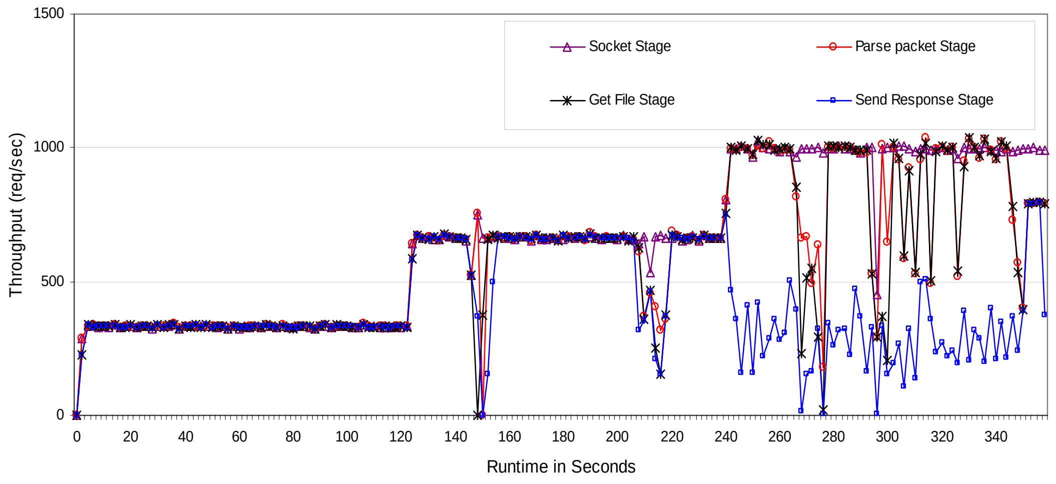


Figure 5-15 Performance of the Stages in the Original SEDA

Finally, our design employs the automatic control theories which implement the fairness management instead of the manually configured fixed control policies. Using this approach, system administrators are only needed to configure the performance target of the whole system. However, in terms of the fixed policy, the original SEDA is required to optimize a set of parameters such as the number of tokens and the thread pool size etc. for each stage. From these experiment results, we can see that when the system is running under the dynamic loading environment with uncertainties, our automatic control mechanism over fairness is able to maintain a stable satisfied performance, meeting the target of the requirement, but the performance of the original SEDA is usually hard to guarantee in terms of the fixed configurations.

As a summary of the above comparison, according to the evaluation on a real-life SEDA-based web server, our hybrid control is an effective approach to providing the expected quality of service to more requests and adaptively reacting to the change of loadings. Compared with the admission control used in the original SEDA, our design exhibits more robustness and provides a fairer service to the incoming requests, especially when the server is running under unpredictable dynamic loading environments. In addition, the designs of the global control framework and the auto-tune stage are able to enhance the quality of the utilization of the system resource and reduce the cost of the manual configuration of the control policies, which significantly simplifies the configuration management of the staged system. As a result, the overall performance of the hybrid control is better than pure the admission control at the source. And the SEDA-based internet application with hybrid control is able to provide a fairer service to high-concurrent requests than the original designs.

Chapter 6 Conclusions and Future Work

6.1 Conclusions of the Thesis

This thesis presents a new design to provide ensured fair service to high-concurrent requests. Based on the SEDA architecture, the presented design makes use of a global control framework to balance loading in multi-queue event-driven systems, high-level adaptive control stages to self tune its own resources, and a selective dropper to filter the long-queuing requests.

Instead of applying complicated control theory and algorithms, experiment results demonstrate that proportional control based pre-compensation model is an effective approach to performance control over multi-stage software systems, and is able to provide faster convergence, better accuracy and more reliable quality of service compared to the manual-configured heuristic control mechanisms. This makes it feasible to build our approach into the SEDA middleware and apply it for a large range of applications.

Compared with the fair service provided by the original SEDA, our design provides a fairer and more robust service for high concurrency with a much lower rejection rate, and it is much easier to achieve the desired overall performance in terms of a global control system, avoiding the complicated configuration of each stage. Both theoretical analyses and experiment results demonstrate that, making use of the combination of resource control and request rate control to manage the fairness of service, is effective for fairness control over web servers with a guaranteed performance even under unpredictable dynamic loading environments.

6.2 Future Work

At this point, “Where to now” is a question placed in front of us. In summary, future work can be grouped into categories as outlined below.

6.2.1 Better Control Strategies

This thesis introduces an innovative control strategy for staged systems, and demonstrates that the proportional control based pre-compensator design is able to provide effective control for the single input single output system. However, a system is generally needed to offer class-based service differentiation. How to combine the request classification technique with the current design is a very interesting research topic for future research. Moreover, our thesis shows that fair service and queuing latency are controlled by the arrival rate and departure rate of the system instead of by control only at the queuing source or the resources. Nonetheless, how to improve the balance between the arrival and departure rate to meet the performance demands will require further study in future work.

6.2.2 Use in Large-Scale Systems

The design presented in this thesis is built on the SEDA architecture. Despite outstanding performance of SEDA, its application is limited on one machine. In theory, the concept of multi-queue event-driven staged network can be extended to distributed systems. And our global control strategy is able to support large scale staged systems rather than a single box. In future work, we intend to implement our design on large scale distributed systems, by coordinating the performance of the machines in the staged pipeline.

6.2.3 Deployment in Single Thread Pool System

The above chapters show that each stage in SEDA can be regarded as a thread-based concurrency system and demonstrate that our P-Control based Pre-Compensator control strategy is able to perform effectively to meet the dynamic demands. Compared to other control approaches such as PI, PD and heuristic control, our design is simple and reliable, as well as providing a fast convergent speed, and meeting a variety of needs of the web applications. This control approach should also be a good strategy for the single thread pool system like Apache. In future, we will implement it into Apache to evaluate its performance.

References

1. T.F Abdelzaher, C Lu. *et al.*: Modeling and Performance Control of Internet Servers. *Proc of the 39th IEEE Conference on Decision and Control Sydney Australia.* (December 2000)
2. T.F Abdelzaher, V Sharma. *et al.*: A Utilization Bound for Aperiodic Tasks and Priority Driven Scheduling. *IEEE TRANSACTION ON COMPUTERS* Vol 53. No.3 (March 2004)
3. T.F Abdelzaher, J. A Stankovic, *et al.*: Feedback performance control in software services. *Proc IEEE control systems magazine* (2003)
4. T.F Abdelzaher, K.G Shin *et al.*: Performance Guarantees for Web Server End-Systems: A Control-Theoretical Approach. *IEEE Transactions on Parallel and Distributed Systems* 13(1):80-96,(2002)
5. K. Abhinav, M. Vishal, *et al* A self-tuning controller for managing the performance of 3-tiered web sites. *In International workshop on Quality of Service (IWQoS)*, Montreal, Canada, pages 47--56. IEEE Computer Society, (June 2004)
6. Apache Web Server http://httpd.apache.org/docs/2.2/
7. K J Astrom *et al*: Adaptive Control (second edition) *Addison- Wesley Publishing Company, Inc.* ISBN: 0-201-55866-1 (1995)
8. V. Beltran, D. Carrera, *et al.*: Evaluation the scalability of java event-driven web servers. *Proc. of International Conference on Parallel Processing (ICPP'04), IEEE.* (2004)
9. C. Benjamin, Kuo and F. Golnaraghi: Automatic Control Systems (8th edition), *Wiley*, ISBN: 0471134767. (2002)
10. S. Bianca, et al.: "Closed versus open system models: Understanding their impact on performance evaluation and system design." Proc of the Networked System Design and Implementation NSDI '06 (2006)
11. J. M. Blanquer, A. Batchelli et al.: Quorum: Flexible quality of service for internet services. Proceedings of the 2nd USENIX Symposium on Networked Systems Design and Implementation (NSDI '05). Boston, MA, USA. (May 2005)
12. J. Carlstrom and R Rom, Application-aware admission control and scheduling in web servers. In IEEE Infocom (2002).
13. H. Chen and P. Mohapatra: Session-based overload control in QoS-aware Webservers. *Proc. of IEEE INFOCOM2002*, pages 516-524. (2002)
14. X. Chen , P. M.,H. Chen.: An admission control scheme for predictable server response time for web accesses. *Proceedings of the 10th international conference on World Wide Web.* (2001)
15. K. Deb, S. Agrawal, A. Pratap, *et al*. A fast elitist non-dominated sorting genetic algorithm for multi-objective optimization: Nsgaii. *In Proceedings of the Parallel Problem Solving from Nature VI Conference* , 16-20 September. Paris, France, pages 849--858. (2000).
16. Y. Diao, F. Eskesen, *et al.*: Generic online optimization of multiple configuration parameters with application to a Database Server. *DSOM 2003: Distributed systems, operations and management. IFIP/IEEE international workshop No14, Heidelberg , ALLEMAGNE.*(20/10/2003)
17. Y. Diao, N. Gandhi, *et al.*: Using MIMO feedback control to enforce policies for interrelated metrics with application to the Apache Web server. *Proc. of the Network Operations and Management Symposium*, Florence, Italy.(2002)
18. Y. Diao, J.L. Hellerstein, *et al.*:Incorportating Cost of Control into the Design of a Load Balancing Controller. *Proc of IEEE RealTime and Embedded Technology and Applications Symposium, pages 376--387. IEEE Computer Society,* (2004)
19. 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)
20. Y. Diao, J.L. Hellerstein., *et al*.: Optimizing quality of service using fuzzy control. *Proc. of the 13th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management*. Springer-Verlag, pages 42--53. (2002)
21. Y. Diao, J.L. Hellerstein., *et al.*: Self-Managing Systems: A Control Theory Foundation *IBM Research Report RC23374(W0410-080)* (October 13.2004)
22. Y. Diao, J.L. Hellerstein., *et al.*:Using fuzzy control to maximize profits in service level management. *IBM System journal* Vol 41, No 3. (2002)

23. R. Doyle, J.F. Chase., *et al.*: Model-Based Resource Provisioning in a Web Service Utility. *Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems (USITS '03),* (Mar. 2003)
24. E. J. Friedman, S. Henderson. : Fairness and efficiency in web server protocols, Prof of ACM SIGMETRICS international conference on Measurement and modeling of computer systems. Pages: 229 – 237 (2003)
25. F Harada, T Ushio, *et al,*: Adaptive Resource Allocation Control with On-Line Search for Fair QoS Level. *Proc of the 10th IEEE Real-Time and Embedded Technology and Applications Symposium(RTAS'04)* (2004)
26. J.L Hellerstein, Challenges in Control Engineering of Computing Systems *IBM Research Report RC23159(W0309-091)* (September 16.2003)
27. J.L Hellerstein, Y Diao, *et al*.: A Framework for Applying Inventory Control to Capacity Management for Utility Computing. *IBM Research Report RC23373(W0410-079)*(October 13.2004)
28. J.L Hellerstein, Y Diao, *et al*.: Applying Control Theory to Computing Systems *IBM Research Report RC23459(W0412-008)* (7 December.2004)
29. J.L Hellerstein, Y Diao, *et al*.: IBM Research Report: applying control theory to computing systems. *Proc. of Computer Science RC23459 (W0412-008).* (2004)
30. J.L Hellerstein, Y Diao, *et al*.: Feedback control of computing systems, *IEEE Press Wiley-Interscience*, ISBN:0-471-26637-X. (2004)
31. D. Henriksson, Y Lu. et al.:Improved Prediction for Web Server Delay Control. *Proc of the 16th Euromicro Conference on Real-Time Systems (ECRTS'04),* Catania, Sicily, Italy, (June 2004)
32. Httperf http://www.hpl.hp.com/research/linux/httperf/docs.php
33. W. LeFebvre. CNN.com: Facing a world crisis. Invited talk at USENIX LISA'01, December 2001.
34. Y Ling, T. M., and X Lin. "Analysis of Optimal Thread Pool Size." (2000)
35. L Ljung,.: System identification: theory for the user 2nd ed. *Upper Saddle River, N.J: Prentice Hall,* ISBN: 0136566952. (1999)
36. X Liu, S. Lui, *et al*.: Online Response Time Optimization of Apache Web Server. *Proc. of the 11th International Workshop on Quality of Service (IWQoS 2003)*, pages 461-478. (2003)
37. C Lu, T.F. Abdelzaher, *et al*.: A feedback control architecture and design methodology for service delay guarantees in web servers. *Technical Report CS-2001-06*, University of Virginia, Department of Computer Science. (2001)
38. C Lu, T.F. Abdelzaher, *et al*.: A Feedback Control Approach for Guaranteeing Relative Delays in Web Servers. *In IEEE Real-Time Technology and Applications Symposium*, pages 51--62,(June 2001)
39. J. M. Maciejowski. Predictive Control with Constraints. *Prentice Hall*, 2002, ISBN: 0-201-39823-0
40. D.A. Menasce: Load testing of web sites. *Proc of IEEE internet computing* (July August 2002)
41. D.A. Menasce and V.A. 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)
42. D.A. Menasce and V.A. F.Almeida :Capacity Planning and Performance Modeling, *Prentice Hall PTR Upper Saddle River,* N.J.07458, ISBN: 0-13-789546-1.(1994)
43. the MathWorks Inc: System Identification Toolbox
44. G. Nutt. : Operating Systems, Third Edition. *Addison Wesley*. ISBN 0-201-77344-9 (2004)
45. A.V. Oppenheim, Schafer R.W., *et al*.: Discrete-time signal processing, *Prentice Hall Signal Processing Series,* ISBN: 0137549202. (1999).
46. G. Pacifici, W. Segmuller, *et al*, Managing the response time for multi-tiered web applications, *IBM, Tech. Rep. RC 23651,* 2005.
47. D. Pendarak, J. Silber and L. Wynter. : Autonomic Management of Stream Processing Applications via Adaptive Bandwidth Control, Proceedings of the 26th IEEE International Conference on Distributed Computing Systems (2006)
48. P. Pradhan, R. Tewari, *et al*. An Observation–based Approach Towards Self-Managing Web Servers. *Workshop on Quality of Service* (2002)
49. Ivy Schmerken, "Can the Market's Systems Keep Up With Electronic Trading?" Wall Street & Technology February 28, 2007
50. L. Sha, X. Liu *et al*.: Queueing Model Based Network Server Performance Control. *Proc of the 23rd IEEE Real-Time Systems Symposium* (2002)

51. SPECweb99 Benchmark, http://www.spec.org/web99/
52. SUN Microsystems INC. New I/O APIs. http://java.sun.com/j2se/1.4.2/docs/guide/nio. (2002)
53. TPC-W: http://www.tpc.org/tpcw/
54. Tuning Garbage Collection with the 1.4.2 Java[tm] Virtual Machine. http://java.sun.com/docs/hotspot/gc1.4.2/
55. 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.)
56. Vivek S. Pai, Peter Druschel, *et al*. Flash: An efficient and portable Web server. *Proc. of the USENIX 1999 Annual Technical Conference.* (1999).
57. M. Welsh: NBIO: Nonblocking I/O for Java http://www.eecs.harvard.edu/~mdw/proj/java-nbio/
58. M. Welsh.: An architecture for highly concurrent, well-conditioned internet services (Thesis). *Computer Science, University of California at Berkeley*. (2002)
59. M. Welsh and D. Culler: Adaptive Overload Control for Busy Internet Servers. *Proc. of the Fifth USENIX Symposium on Internet Technologies and Systems* (2003)
60. M. Welsh and D. Culler: Virtualization considered harmful: OS design directions for well-conditioned Services. *Proc. of the 8th Workshop on Hot Topics in Operating Systems (HotOS VIII).* (2001)
61. 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)
62. D. Xu,: Performance study and dynamic optimization design for thread pool systems, Masters thesis, *Iowa State University.* (01 July 2004)
63. J.Zhou, T.Yang.: Selective early request termination for busy internet services, Proceedings of the 15th international conference on World Wide Web (www2006) Edinburgh, Scotland (2006).
64. 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).
65. X. Zhou, C Xu, : Harmonic Proportional Bandwidth Allocation and Scheduling for Service Differentiation on Streaming Servers. *IEEE Transactions on Parallel and Distributed Systems*. Vol. 15 No.9 (September 2004)
66. http://www.internetworldstats.com/stats.htm