NOTE TO USERS

This reproduction is the best copy available.

An Architecture-based Self-Adaptive Framework for Performance and Reliability Improvement

by

Xu Zhang

A thesis submitted to the Faculty of Graduate Studies and Research

in partial fulfillment of the degree requirements of

Master of Applied Science in Electrical Engineering

Ottawa-Carleton Institute for Electrical and Computer Engineering (OCIECE)
Department of Systems and Computer Engineering
Carleton University

Ottawa, Ontario, Canada, K1S 5B6

January 2010

Your file Votre référence
ISBN: 978-0-494-63816-3
Our file Notre référence
ISBN: 978-0-494-63816-3

Abstract

Distributed and concurrent systems have become common in enterprises today, and the complexity of these systems has increased dramatically. A self-adaptive system is a preferred approach because it can acclimate to a dynamically changing environment. To achieve this goal, this thesis proposes a Self-Adaptive Framework for Concurrency Architecture (SAFCA). SAFCA can include multiple concurrency architectural alternatives and adapt to the appropriate architecture based on changes in the environment. With autonomic control, SAFCA can handle bursty workloads by invoking another architectural alternative at run time instead of statically configured to accommodate the peak demands. SAFCA can also improve reliability if there is a failure in the system. Experiment results demonstrate that the performance of SAFCA is better than systems using a stand-alone concurrency architecture and the reliability of the system using SAFCA is also improved.

To Shan Xie

Acknowledgements

Many thanks go first to my supervisor, Professor Chung-Horng Lung, for his invaluable guidance, thoughtful advice, and continuous support.

I also like to thank Professor Greg Franks for his comments and support on LQNs and performance analysis for concurrency systems.

I would also like to thank NSERC, Canada, for partially providing financial support for this research. I also like to express thanks to MITACS, Canada and Cistel for their financial support for the preliminary study.

In addition, and as always, I am very grateful to my friends and my family for their direct and indirect contributions to this thesis. Special thanks go to my parents for their untiring encouragement.

Table of Contents

List of Tables

List of Figures

Acronyms

DTC	Dynamic-Thread-Creation
F-ORB	Forward Object Request Broker
H-ORB	Handle-Driven Object Request Broker
HS/HA	Half-Synchronous/Half-Asynchronous
LFs	Leader/Followers
LQN	Layered Queuing Network
ORB	Object Request Broker
SAFCA	Self-Adaptive Framework for Concurrency Architectures
SAFCA-A	SAFCA using Arrival Rate and Response Time Based Self-Adaptive Policy A
SAFCA-B	SAFCA using Arrival Rate and Response Time Based Self-Adaptive Policy B
SAFCA-Q	SAFCA using Queue Length Based Self-Adaptive Policy
SAFCA-R	SAFCA using Reliability Self-Adaptive Policy
TCP	Transmission Control Protocol
UDP	User Datagram Protocol

Chapter 1 Introduction

In recent years, with new software engineering theories and practices pushing forward and arousing general interest, both the average complexity of computer systems and the number of computing devices in use have been increasing dramatically [EMA06]. As a result, IT personnel have had to shoulder the burden of time-consuming (and therefore expensive) supporting tasks such as configuration, maintenance and system performance evaluation [Kep03]. Further, manual control of a large distributed computing system is invariably prone to errors.

The goal of autonomic computing, initiated by IBM in 2001 [EMA06], is to define rules for a system for controlling its behaviour so that the system regulates its actions to automatically configure, heal, protect, and optimize itself [Kep03]. Many research organizations and IT companies have launched research projects related to autonomic computing recently [Mul06], including in the area of distributed and concurrent applications, due to their high complexity.

Distributed and concurrent systems have become common for enterprises these days [Fra96], with multi-tier systems incorporating a web-server, application-server, database-server and others [Hag09]. However, concurrent systems contain several architectural configurations that can impact their performance. Furthermore, the Internet is known for

its dynamic nature. For instance, under normal circumstances, the growing server farm of the Cable News Network (CNN) website, following a distributed system model, is more than adequate to handle regular traffic demands. However, during special events, the server farm may become unavailable due to unprecedented demands [Wel03]. Another example is the electronic stock trading sites: unexpected increases in volume and heavy sell-off can also result in periods of down-time. This is another sign of how vulnerable the market is to systems overloads and data backlogs [Sch07]. Therefore, servers must adapt to a sudden change in the environment quickly and since manual server reconfiguration has been shown to be inadequate [IBM06], self-adaptive solutions are better suited for such problems.

1.1 Motivation

As stated earlier, distributed and concurrent systems have become common today. Software architecture plays a critical role in different phases of software development. For distributed and concurrent systems, there is often more than one software architecture available and many configurations exist. Those architectures have the same functional requirements, but they have different quality impact and performance results that are highly coupled with network traffic load, operating systems and hardware [Lun06]. Various factors, such as workloads, system resources, contentions, complex interactions between the application and the kernel, and increasing hardware complexity like threading and parallelism, can affect performance. During run time, all these factors are constantly changing and it is hard to estimate demands and select the architecture and

suitable associated configurations. Manual selection of the best software architecture is ineffective in this situation.

Furthermore, many systems today are configured initially to handle worst-case scenarios; in other words, systems are over-provisioned. The problem with this approach is that many resources will be wasted in normal condition scenarios. The aim of autonomic computing is to provide a solution that runs services with minimum cost, capable of scaling up and scaling down the system resources responding to dynamic demands. This requires system elasticity, in terms of allocating or using resources as they are needed.

In the initial study [Zha10], the performance modeling and analysis for three concurrency architectures, based on the Layered Queuing Network (LQN) [Fra96], were carried out by using LQN simulation tools. It was demonstrated in the preliminary study that the performance of two critical concurrency architectures are different, depending on the network traffic loads and system resources. Therefore, incorporating the self-adaptive capability into a distributed and concurrent system [Lun07] (for example, a multi-tier system) is desired to increase system performance.

Another motivation of the thesis is related to reliability improvement. Even with ever-improving development processes and tools, modern software is still not bug-free because some bugs that are caused by race conditions, resource leaks, etc. [Can04] are difficult to find and resolve. Moreover, up to 80% of bugs [Can04] manifest during run time and have no fix available at the time. Self-adaptation from one architectural alternative to another one at runtime due to failures could improve reliability [Whi09].

The idea of switching to another alternative is that similar errors may recur using the same architecture or design even the same system is restarted. By switching to another architecture or design, the chance that the same errors will happen again is smaller. Debugging of the original design can then be conducted while the other alternative is running.

To summarize, one motivation for this thesis is to improve or avoid degrading of performance of multi-tier systems under dynamic or bursty workloads. Another motivation is to improve the reliability of multi-tier systems that are susceptible to partial system failure. Thus, this thesis will study a mechanism to develop a self-adaptive framework that supports different concurrency architectures, adopts autonomic computing techniques [Bru09], and adapts to changing environments. In addition, the solution will enhance both performance and reliability, and must be practically implementable for real problems.

1.2 Contributions

This thesis is involved in multiple research areas, including software architecture, autonomic computing, software performance engineering, reliability, and empirical study. The contributions of this thesis cover the following areas:

i. Software Architecture: Developed a new approach and a mechanism, Self-Adaptive Frame work for Concurrency Architectures (SAFCA), which supports multiple concurrency architectures and self-adaptation at the architecture level. Moreover, new concurrency architectures can be easily inserted into SAFCA.

ii. Autonomic Computing: Developed various control mechanisms to facilitate self-adaptation of an autonomic system. Intensive experiments have been conducted to evaluate the control mechanisms and the impact of those mechanisms.

iii. Performance: The control mechanism increased the performance of multi-tier architecture under different workloads, through a self-adaptive means.

iv. Reliability: The mechanism also improved the reliability of multi-tier architecture under single architecture failure through a self-adaptive means.

Parts of this thesis have appeared in the following publications:

i. Lung C.-H., Zhang X., Franks G., Zaman M., "Towards A Performance-Oriented Self-Adaptive System", *Proc. of the 6th International Workshop on System and Software Architectures*, June 2007.

ii. Lung C.-H., and Zhang X., "Treating Software Design as a Service to Manage Complexity", *Proc. of IEEE International Conference on Service-Oriented Computing and Applications*, Dec 2009.

iii. Zhang X., Lung C.-H., and Franks G., "Towards Architecture-based Autonomic Software Performance Engineering", to appear in the *Proc. Of the 4th Conference on Software Architectures*, March 2010.

To the best of the author's knowledge, no existing work covers all these areas.

1.3 Thesis Organization

This thesis is organized as follows: Chapter 2 provides the background information and current state of research on the topics of autonomic computing, self-adaptive system, and concurrency architectures. Chapter 3 describes the self-adaptive framework, SAFCA,

proposed in this work. Chapter 4 presents the experiment used to compare the performance of the self-adaptive concurrency architecture with other stand-alone concurrency architecture. The performance and reliability results are then analyzed. Finally, Chapter 5 concludes with the contribution of this thesis and addresses future research areas.

Chapter 2 Background and Literature Review

This chapter provides the background information on autonomic computing, the current state of research on self-adaptive systems, and the various existing concurrency architectures.

2.1 Autonomic Computing

A significant limiting factor for the further development of modern distributed computing systems is that their complexity is becoming too large to manage effectively. Manual control of a large distributed computing system is time-consuming, expensive, prone to errors, and tends to increase very quickly as the size of the system grows. The goal of the "Autonomic Computing Initiative", started by IBM in 2001, is to define rules for a system for controlling its behaviour so that the system itself regulates its own actions; much like the autonomic nervous system of an animal regulates its own actions, such as breathing, without conscious effort [EMA06].

The following four functional areas have been defined by IBM [IBM06]:

i. Self-Configuration: automatic configuration of components;

ii. Self-Healing[Shi06]: automatic discovery, and correction of faults;

iii. Self-Optimization: automatic monitoring and control of resources to ensure the optimal functioning with respect to the defined requirements;

iv. Self-Protection: proactive identification and protection from arbitrary attacks.

This thesis focuses on two items: self-optimization and self-healing.

IBM's blueprint for autonomic computing defines autonomic managers that are used to manage software or hardware resources. The autonomic manager is a component that implements the control loop, including four parts [IBM06], as shown in Figure 2.1:

i. The monitor function collects the relevant information from the system environment that reflects the current state of the system. Two engineering questions that need to be answered in the monitor function are: What is the required sampling rate? Does it provide sufficient information for system state analysis?

ii. The analyze function is to analyze the information collected by the monitor and determine if anything needs to be changed. Many approaches can be used to structure and analyze the raw data (e.g., using models, theories, and rules). In the analyze function, some relevant questions that need to be answered are: How is the current state of the system inferred? How much information on the system's past states may be needed? What data needs to be archived?

iii. The plan function is to create a plan, or sequence of actions, that specifies the necessary changes. A decision must be made about how to adapt the system in order to reach a desirable state.

iv. The execute function is to implement the decision. The system must perform the actions. An important question that arises is: How can the adaption be performed safely and quickly?

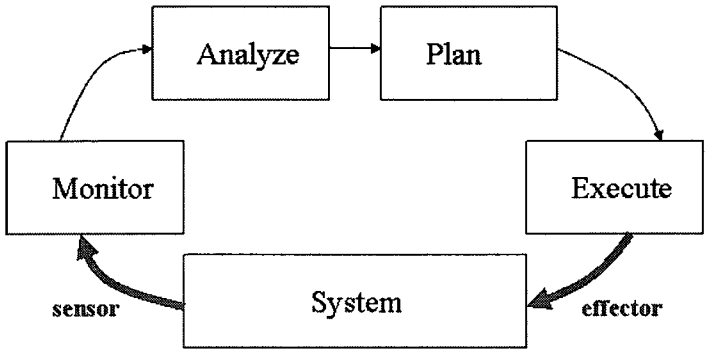

Figure 2.1: Autonomic Computing Control Loop [IBM06]

When these functions are automated, an intelligent control loop is formed.

2.2 Self-Adaptive Systems

The early inspiration of the self-adaptive concept is based on Darwinian Evolution [Bec08]. As natural organisms adapt to their environment, computing systems also strive to adapt to their changing environment.

Self-adaptive systems have been studied in different research areas, such as distributed systems [Mi09], Web systems [Wel03], middleware [Abd02], component-based architecture [Hag09], and software architecture [Geo02]. The system's ability to change its behaviour in response to the environment in the form of self-adaptation has become one of the most promising research directions. The "self" prefix in the name "self-adaptive" means that the system autonomously decides how to adapt or reorganize itself to accommodate changes in environments. "Autonomously" means without or with minimal interference.

Self-adaptive systems have been studied within different research areas. Other research communities that have also investigated self-adaptation and feedback from their

own perspectives are even more diverse. In this literature review, each work has incorporated the self-adaptive concept into a specific application area, with each work focusing on one aspect of the autonomic control loop: "collect", "analyze", "decide", and "act" [Dob06].

Each work in the following literature review is discussed within the following structural framework:

i. What area the self-adaptive concept is being applied. Some areas are web systems, distributed system, middleware, etc.

ii. The information being collected can be the queue length of a system, the response time of the system, the system utilization, etc.

iii. How the information is collected. The collection approach can be measurement-based or prediction-based. Self-adaptive systems using the measurement collection approach are reactive systems: they detect a change in the environment and react to it. Self-adaptive systems using the prediction approach are proactive systems: they predict a change ahead of time and react to it.

iv. The action taken is based on the information. The action can limit the arriving requests, adding additional hardware/software resources, switching to a different mode of operation, reconnecting component-based architecture, etc.

In [Ben09], virtual machine instances are used to realize multi-tier client-server services (multi-tier: web server, middleware server, database server). The monitor component measures the response time between request and response, for TCP/IP sessions. This information is used to identify performance bottlenecks in the system and

remediating actions are taken. Actions include adding extra virtual machine instances and/or increasing resources, for instance.

[Che09] adds an anticipatory approach to self-adaptive frameworks for a web-based client-server system that conforms to an N-tier style. The resource prediction approach is based on historical information as well as the temporal horizon of the decisions (meaning, when decisions are made periodically, the time between now and the next decision point is call temporal horizon), and takes uncertainty in the available information into consideration. The predicted information is used in the implementation of the self-adaptive system using autonomic computing concept. Actions to be taken include adding or removing system hardware resources (adding additional servers).

The work in [Mi09] is conducted in the area of multi-tier architectures and online Data Stream Management Systems. The goal is to improve the system response time by taking advantage of the autocorrelation of service times. When the system becomes overloaded, jobs that require long service time are dropped. The work claims that one can improve response time drastically with only a small increase in packet loss. The work assumes there are only two service times for each request, one long and one short. The jumping probability between them is defined by the author.

[Gra09] incorporate utility functions into autonomic application. The case study used was the FTP download, and the work monitors the socket throughput, where the first five statistical moments of the throughput samples were determined. Those moments are used by the utility function to determine a single utility value. The utility value can be used to

accurately determine the system's performance. However, this work does not mention what actions can be taken to improve the utility value if the value is not satisfactory.

[Hag09] is focused on the load-balancing issue in a multi-tier clustered system. An old legacy software piece that does not support autonomic computing is wrapped in components in order to be incorporated into an autonomic computing system. The system monitors the CPU utilization on servers. If the utilization on one server becomes too high, workload is diverted away from it. Thus, the system supports self-optimization and self-repair.

[Reh09] uses the concept of self-adaptation in the area of intrusion detection. In this work, an intruder has an ultimate goal, such as taking over the server. To achieve this goal, tasks need to be conducted (in the paper, each task is an "attack"). The tasks are organized in a hierarchical structure, as some tasks cannot be carried out until some other tasks are finished, while other tasks can be performed simultaneously. Thus, this work uses a tree with the goal at the root and the tasks as leafs. The behaviour of each user/process/connection is monitored and compared against this tree and those "matching" the tree are treated as attackers. Actions such as logging off the user, killing the process, or cutting-off the connection are taken.

[Whi09] presents a technique to self-heal complex service-oriented architectures. The case study in this work is focused on an e-commerce application. Each component of the application is placed in a container using a composition feature model. This technique identifies feature failures by using event stream processing, which is to run queries

against the application's event data. If a feature failure is identified, the feature model derives an alternate set of feature components, excluding the failed feature.

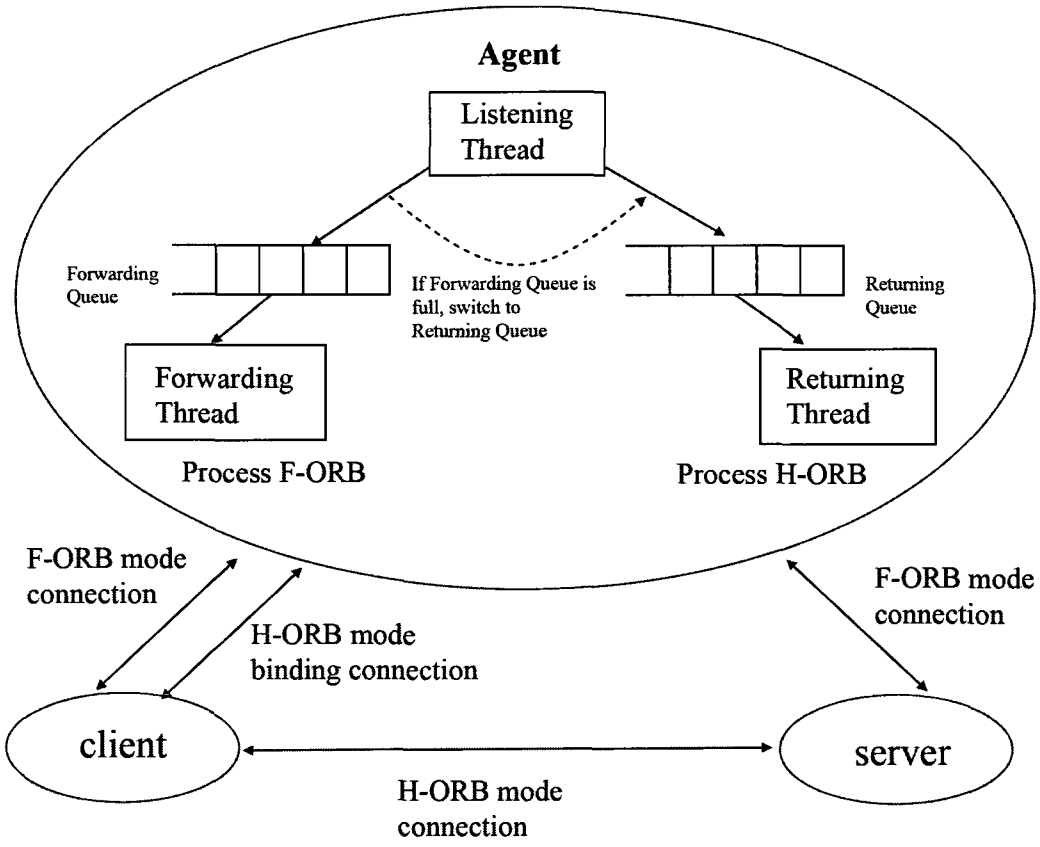

Figure 2.2: Adaptive ORB [Maj03]

[Abd02][Maj03] are in the area of CORBA-Based Client-Server Architectures. They include an adaptive approach which can dynamically switch (as shown in Figure 2.2) from an F-ORB (Forward-Object Request Broker) based mode of operation to an H-ORB (Handle driven-Object Request Broker) mode of operation. In the F-ORB mode, client and server communication are all through an agent; this mode is efficient under light workload, but inefficient under heavy workload since the agent becomes the performance

bottleneck. In the H-ORB mode, the agent binds the client with the server, so the client and the server communicate with each other directly. This mode is efficient under heavy workload. However the binding process requires extra time and may not be efficient under a light workload. Therefore, the adaptive approach dynamically switches between the two modes, depending on the workload. In our work, instead of switching the entire system mode of operation, the architecture is switched on the server side alone.

[Liu08] provides an autonomic middleware solution for QoS controls. The system monitors the response time, throughput and CPU utilization. The QoS control uses a token bucket implementation to adapt to the environment.

The thread pool in [Wel02] is used for email service. The monitor observes a history of response times in the email server. As shown in Figure 2.3, a controller acts on the response time information provided by the monitor to control a token bucket. If the response time becomes unacceptable, the token bucket disallows any new additional request from entering the system.

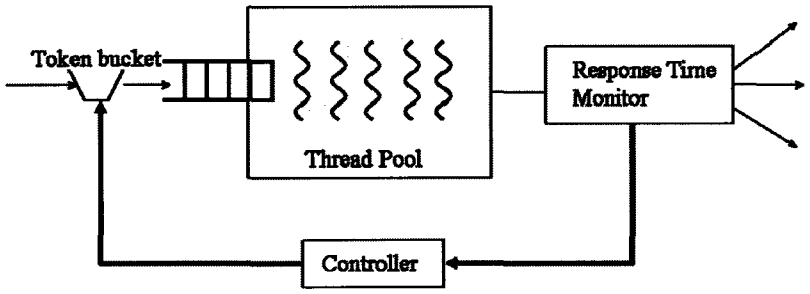

Figure 2.3: Response Time Controller Design [Wel02].

Threshold control [Urg05][Wel03] is widely used in control systems because of its simplicity: monitoring the value of variables. When the value reaches a certain threshold,

then the controller performs certain actions. This work will not go into detail about control-system theories since they are not the focus of this thesis.

2.3 Existing Concurrency Architectures

Concurrent programming has been widely used in many applications. The reasons why concurrent programming is useful and can improve an application's overall performance are as follows [Koc04]:

i. Assume each task in a computing system consists of an I/O bound operation and a CPU bound operation. If tasks are run in sequence, then while one task is doing an I/O bound operation, the CPU cycles are wasted since other tasks cannot use it. In contrast, tasks running concurrently can use the CPU while another task is waiting for its I/O to finish. Thus, concurrent tasks can run faster than tasks running in sequence [Wel00]. Additionally, to take advantage of multi-processor hardware, tasks must run in parallel.

ii. Concurrency also enhances fairness and availability between tasks that have a long running time and between those tasks that have a short running time. In contrast, if tasks are run sequentially, the longer task arrives first, and the shorter task has to wait for the longer task to complete [Lea99].

iii. Concurrency can also increase the application responsiveness. One high priority thread can handle user requests while another thread can run concurrently to perform the necessary operations. Thus, long operations do not freeze the application's screen. [Shi00].

iv. Concurrency can also provide a more appropriate structure for systems that have to control multiple activities and handle multiple events. Instead of using a single task/thread to perform all the functions, concurrency can model the environment in an object oriented fashion.

There are several different known concurrency architectures reported in the literature. The following describes some typical concurrency architectures.

2.3.1 Dynamic-thread-creation

The dynamic-thread-creation (DTC) concurrency architecture is a thread-per-request architecture. It is commonly used in multi-threaded programming, especially for server applications. DTC spawns a thread for each newly arrived request, as shown in Figure 2.4. A thread dies after processing its request [Wel00].

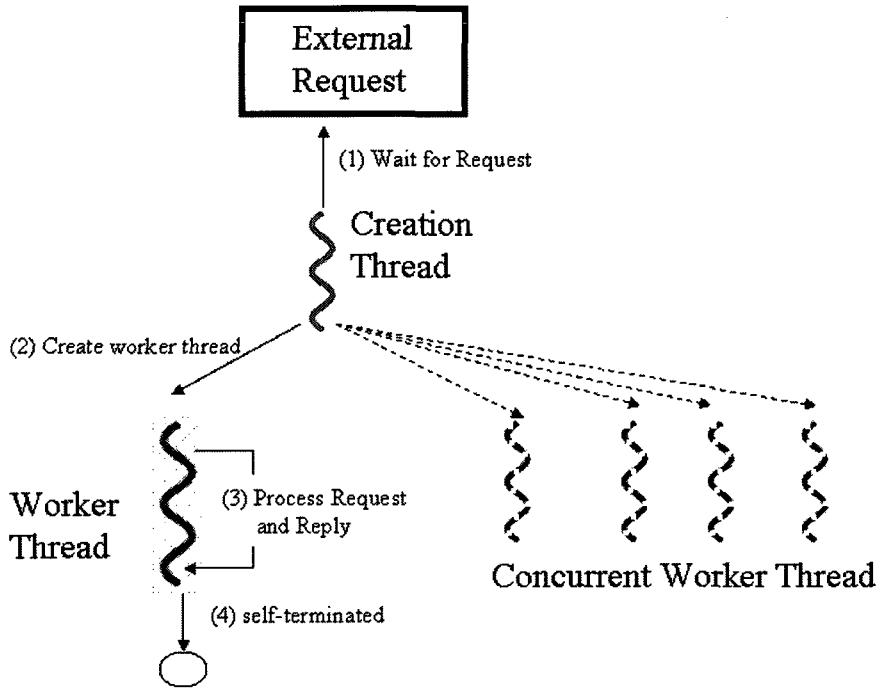

Figure 2.4: Dynamic-Thread-Creation

Because programmers can write straight-line code and rely on the operating system or thread library to handle all thread creation and destruction details, DTC is relatively easy to program. However, each thread has to be dynamically created for each new request, there is an overhead associated with it.

When the number of requests increases, a large number of threads may be created in the system, since each accepted request occupies one thread. Once the number of threads crosses a threshold (this threshold depends on to the system maximum capacity), DTC generates a huge amount of overheads, for dynamic thread management and thread context-switching. As a result, the performance of the system begins to degrade [Har05].

2.3.2 Thread Pool

In contrast to the DTC concurrency architecture, a number of threads are allocated to a thread pool during initialization and no additional threads will be created at runtime [Wel00]. Each thread is engineered not to die when its task is completed. Instead, it returns to the thread pool to wait for the next request. Since a thread pool can reduce overheads resulting from thread creation and destruction, a thread pool shows better performance and stability than does creating a new thread for each task. The thread pool model is deployed in a large number of popular computing system designs.

However, in a thread-based concurrency architecture, when the number of requests is beyond the thread pool limitation, no extra threads are allowed to be added into the thread pool. The system throughput becomes limited by the pool size, since additional requests thus cannot be processed until threads have been released from the current request process. In a typical thread-based system, a request occupies one thread until the end of

its operation. When there is no available thread in the pool, all new incoming requests need to stay in the queue. If the queue becomes full, any additional requests are dropped. Hence, if there is a high-burst of requests, or if there are any requests that requires a longer processing time, a large number of requests then suffer a long waiting time or even be discarded [Har05].

2.3.3 Leader/Followers

Leader/Followers (LFs) concurrency architecture is structured to minimize the concurrency-related overhead and to prevent race conditions [Sch00b]. In this architecture, the leader is a thread that waits for a request. There is only one leader at a time in this architecture. The followers are other threads in a thread pool that are waiting for their turn to become the leader.

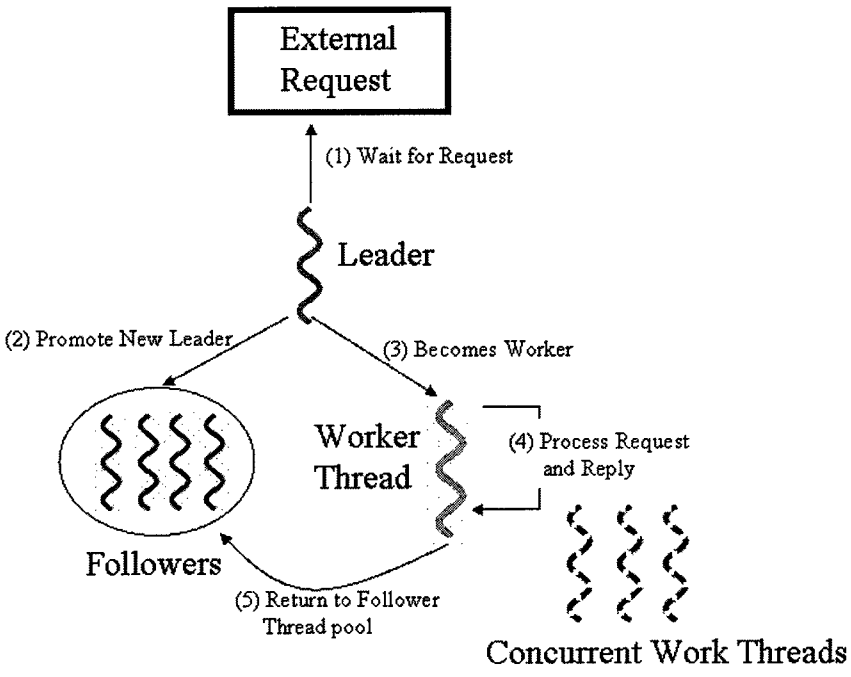

Figure 2.5: Leader/Followers

As shown in Figure 2.5, after the current leader thread receives a request, it promotes a follower thread to become the new leader and then itself becomes a worker thread to process the request. During this period, the former leader and the new leader thread can execute concurrently. There could also be other worker threads that are running concurrently. Once a worker thread has finished the work, it becomes a follower thread and returns to the thread pool.

LFs has shortcomings in terms of implementation complexity and lack of flexibility. Additional implementation complexity comes from the fact that the state of each thread changes frequently, concurrently, and in an unpredictable order. Therefore, ensuring operation atomicity creates additional complexity in the implementation. In addition, there is no queue in this architecture, it is harder to discard or reorder requests already in the system, leading to a lack of flexibility [Sch00a].

2.3.4 Producer-Consumer

In the producer-consumer concurrency architecture, the producer is a thread that, in a loop, generates requests to be processed, as shown in Figure 2.6. The consumer is also a thread that, in a loop, takes those requests and acts on them. Data queues are used to communicate the requests between the producer and the consumer. A mechanism (synchronization, mutex, and etc.) is needed for the producer/consumer to know when it is safe to attempt to write or read data from the queue [Oak04].

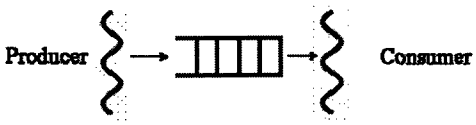

Figure 2.6: Producer and Consumer.

The advantages of the producer-consumer concurrency architectures are [Oak04]:

i. Decoupling of processes that produce and consume data at different rates.

ii. Addition of data buffering between producer and consumer loops to reduce data loss.

iii. Flexible design for different service requirement. Multiple producers serviced by a single consumer, or multiple consumers serving a single producer.

iv. Separation of concerns. The producer can be modified or replaced without any changes need to be made to the consumer, and vice versa.

2.3.5 Half-Sync/Half-Async

The Half-Sync/Half-Async (HS/HA) concurrency architecture evolves from the Producer-Consumer and the thread pool concurrency architecture. In order to simplify programming without unduly reducing performance, the HS/HA decouples asynchronous and synchronous service processing in concurrent systems.

There are two intercommunicating layers and one queuing layer in this concurrency architecture, as shown in Figure 2.7. One layer is for asynchronous service processing and another layer is for synchronous service processing. The asynchronous layer or thread has high priority, which deals with newly arrived requests from external input, and a thread pool of worker threads in the synchronous layer that process the requests.

The need for the HS/HA concurrency architecture arises from performance-sensitive concurrent applications. One example of such an application is the telecommunication switching systems that perform a mixture of synchronous and asynchronous processing [Sch00a].

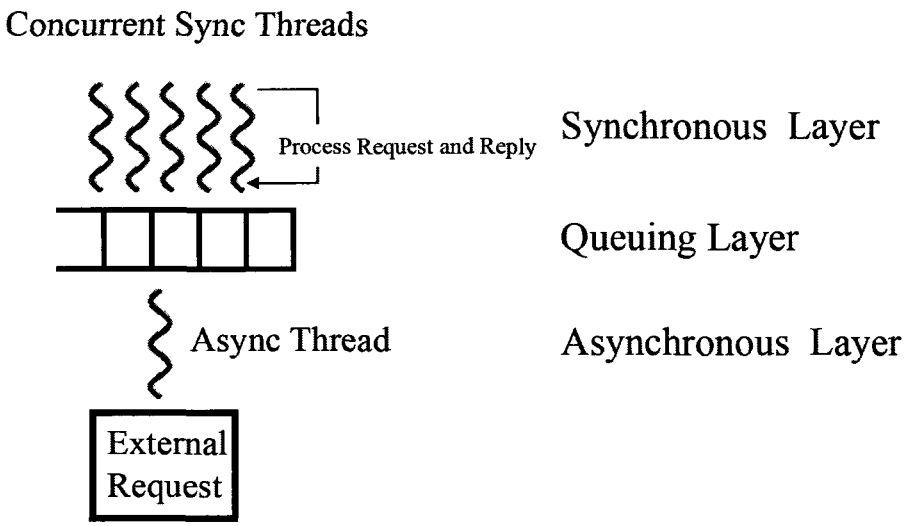

Figure 2.7: Half-Sync/Half-Async

External input first arrives at the asynchronous layer. After this layer processes the input, the input is stored in a queue in the queuing layer. The function of the queuing layer is to buffer input from the synchronous layer to the synchronous layer and inform the synchronous layer that input is now available. Worker threads in the synchronous layer retrieve input from the queue and process the input further. In this concurrency architecture, layers are independent from each other and can perform operations concurrently. For example, the synchronous layer does not have to deal with external event sources.

There are some limitations of HS/HA. For instance, a boundary crossing penalty that is the context switching, synchronization and data copying overhead appear during data transfer between layers. Furthermore, debugging and testing is complex, because it is difficult to single-step through the code execution [Sch95].

2.4 Summary

Thread pool is a popular concurrency architecture on which LFs and HS/HA are based. DTC is a competing concurrency architecture. The main difference between these two architectures is that, in LFs, a thread pool creates a number of threads which stay in the system before any request arrives, while DTC spawns threads as requests arrive and those threads die after they are finished with the request.

The size of the thread pool needs to be pre-configured. If the workload is light and the pool size is fixed, only a few threads among the created threads in the pool are used. The unused threads represent wasted system resources; thus, the system utilization of the thread pool is low under a light workload. If the workload is high, once all pre-created threads are used, new requests have to wait in the queue, and the size of the thread pool becomes the bottleneck of the throughput.

In other words, setting the thread pool size too low results in underutilization of system resources, while setting the size too high leads to performance degradation due to increasing overhead of the operating system (scheduling and aggregate memory footprint) [Wel00][Wel03]. In the HS/HA example (which is based on a thread pool), the more worker threads try to access the same queue, the higher the overhead for thread synchronizing on the queue. For example, imagine a warehouse where there is only one door (see Figure 2.8) and workers have to move containers out of the warehouse. Initially, only a few workers are working. Adding more workers will speed up the moving process. However, when the number of workers is increased to a certain

threshold, too many workers will be forced to wait at the door, since the door cannot allow so many workers to pass at the same time. Thus, the moving speed will slow down.

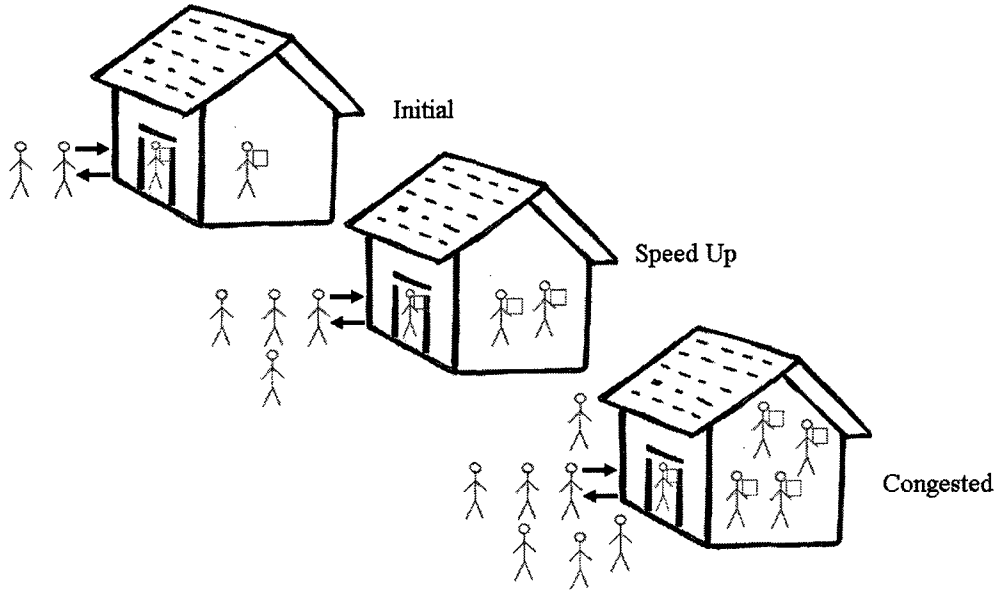

Figure 2.8: Movers Example.

The pool size depends on each worker thread's latency of I/O operations, time cost of CPU operations, thread scheduling, synchronization overhead, and the arrival rate of the network during different time period. All of these parameters are very difficult to determine. However, while DTC does not have the problem of defining a fixed thread pool size, it can have thread-creation/destruction overhead, once there are many threads in the system. In other words, the performance bottleneck for DTC depends on the operating system, whereas the thread pool bottleneck is the number of threads in the pool.

For these reasons, a dynamic self-adaptive approach is the more appropriate solution. For example, a system can implement multiple concurrency architectures. Using autonomic control mechanism, under a certain condition, one concurrency architecture is

used. Under a different condition, the controller selects and switches to another concurrency architecture.

Chapter 3 Self-Adaptive Framework for Concurrency Architectures

The Self-Adaptive Framework for Concurrency Architectures (SAFCA) is to support adaptation at the architecture level during runtime. The framework consists of several components. This chapter discusses each of the components individually and presents the integration to form the framework.

3.1 Problem Definition

Consider a system that implements concurrency, which receives requests from client(s) and processes them concurrently. Two problems have been investigated. One of the problems studied in this work stems from the fact that, under different request arrival rates, different concurrency architectures have different response times, packet loss ratios, and software resource utilizations (such as thread utilization), the framework adapts itself for performance improvement. The other problem studied focuses on self-healing, i.e. the system adapts at the architecture level in event of a failure of one of the concurrency architectures.

The goal of this work is to develop a self-adaptive framework so that: 1) when an overloading request burst occurs, it can achieve acceptable response time and decrease the packet loss ratio; 2) the software resource (for example, the number of threads)

utilization under normal load condition can be maximized; 3) when one of the concurrency architectures breaks down, another concurrency architecture takes over, and; 4) the self-adaptive framework must be practical in the sense that it can be easily implemented.

3.2 Problem Domain

As shown in Section 2.3, multi-tier systems are very popular Web systems [Hag09] [Ben09] [Wel03] [Sch07]. There is usually a client tier, a server tier, and a database tier, as illustrated in Figure 3.1.

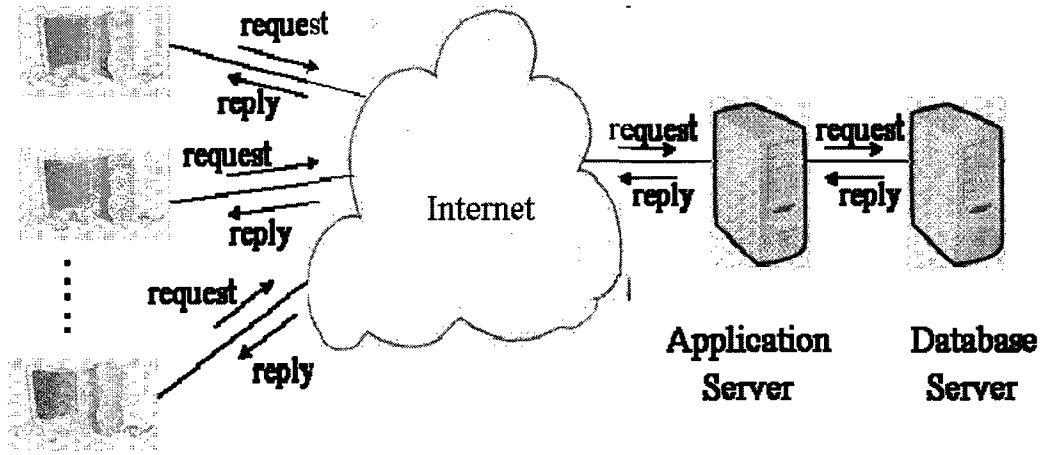

Figure 3.1: Multi-tier System

For each request, the application server has to carry out some CPU-bound operations and some I/O bound operations. CPU-bound operations can be mathematical calculations, cache-lookup, and etc. I/O bound operations include sending requests to a database

server via a network interface card, and waiting for replies. During an I/O operation, CPU cycles are not consumed. As shown in Figure 3.2, each request arriving in the application server requires some CPU-bound operation first (shaded area), then some I/O-bound operation (white area), and finally after some CPU-bound operations, a reply message is sent via an I/O operation. Since the CPU is not used during I/O-bound operations, CPU operations for other requests can be carried out. This significantly improves the system response time, CPU utilization, and throughput and is the main motivation for using concurrency.

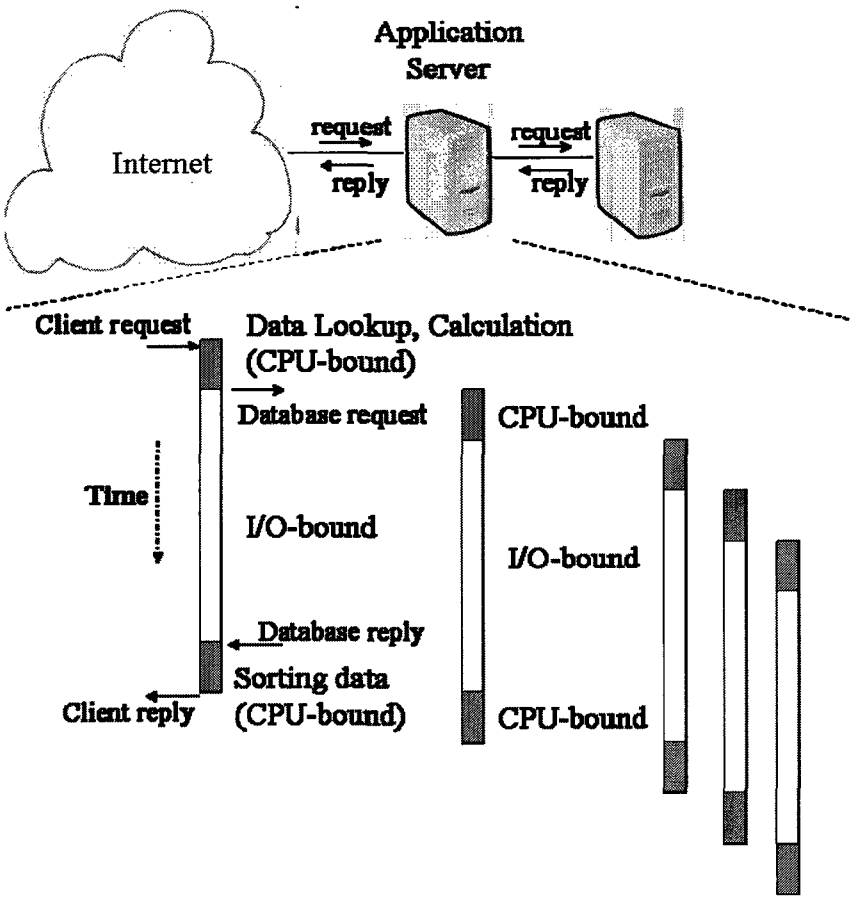

Figure 3.2: Concurrency in Application Server

In contrast, each request cannot be processed until the previous request is completely finished if there is only one flow of execution. This increases response time drastically. Because the CPU becomes idle for long periods of time, the CPU utilization will be low and throughput will also be limited.

Threading is a common strategy for handling concurrency in modern systems. With threading, programmers can write sequential programs and rely on the operating system to overlap computation and I/O operations by transparently switching between threads.

Several different concurrency architectures have been described in Section 2.3, all implemented with threads. As explained previously, the problem is that each concurrency architecture has its weaknesses under different workload conditions. For this reason, this thesis proposes a self-adaptive framework that performs well under different workload conditions.

3.3 Overview of the Framework

The self-adaptive framework proposed in this work implements an autonomic feedback loop. This section presents the software architecture of the self-adaptive framework and defines the function of the architectural components. The self-adaptive framework is implemented with Java, and each component is implemented as a Java thread.

As shown in Figure 3.3, a feedback control loop is implemented in the framework. Some components, the monitor, the analyzer, and the planner, in an autonomic control loop (depicted Figure 2.1) are merged together to reduce inter-component communication overhead. The *Decider* component implements the Monitor, Analyze, and Plan functions

of an autonomic control loop. The *Decider* gathers information regarding the queue length, response time, arrival rate, and number of threads currently running (i.e., the number of threads that have received a request but have not sent a reply). It then computes statistical-average-quantities such as average queue length, average response time, and etc. Based on collected and calculated information, the *Decider* decides if any action should be taken. For example, if the queue length exceeds a threshold, the *Decider* notifies the *Executor* component that the current workload condition is heavy. The reason why the *Executor* component is introduced is to reduce the burden of the *Decider*: so that the *Decider* can keep monitoring, while the *Executor* instructs the *Architecture Manager* to put new requests into the appropriate concurrency architecture queue.

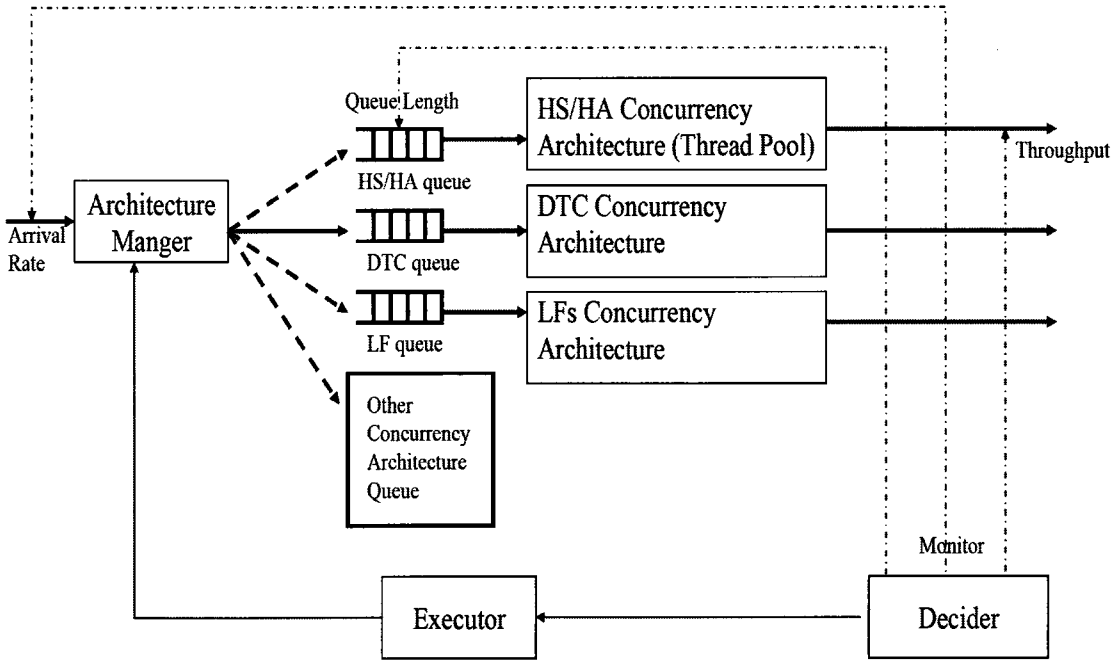

Figure 3.3: Self-Adaptive Framework for Concurrency Architectures Overview

SAFCA is applied to two areas in this thesis: bursty workloads and some architecture-specific failures. Under a sudden burst of requests, SAFCA allocates additional software resources (for example, threads) to meet the sudden surge in demand. After the burst has passed, those resources are freed. In terms of reliability, another concurrency architecture can take over when one concurrency architecture fails (for example, deadlock). SAFCA in this work is implemented using Java programming language. Java was chosen because it has comprehensive built-in support for writing multithreaded programs [Hyd99]. Java also provides strong networking capability.

3.3.1 Bursty Workload Solution

Figure 3.4 describes one type of self-adaptive policy for illustration. As demonstrated in Figure 3.4 (a), self-adaptive mechanism has not been triggered yet. The *Architecture Manager* puts new requests in the initial concurrency architecture queue. In Figure 3.4 (b), when the workload reaches the current configuration capacity limit, the *Architecture Manager* takes action. One sign of work overload is that the queue becomes full. When this occurs, the *Architecture Manager* sends any new requests to a different concurrency architecture queue. The previous concurrency architecture continues to work on the existing requests in its own queue. As depicted in Figure 3.4 (c), once the initial concurrency architecture has finished its work (meaning, the queue length has fallen below a threshold), the *Architecture Manager* sends new requests back to the initial queue.

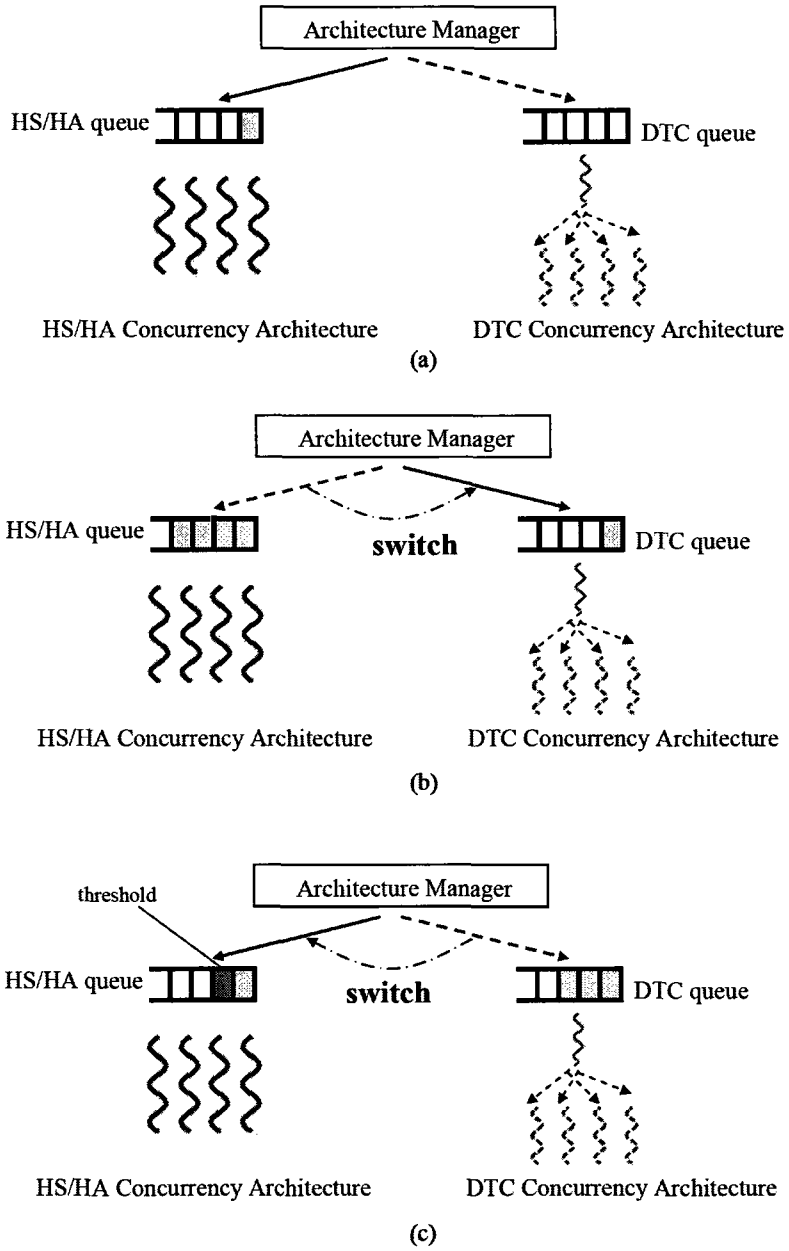

Figure 3.4: Queue Length-based Self-Adaptive Policy

Another trigger to self-adaptive is based on arrival rate and average response time. Arrival rate alone cannot indicate if the system has reached its capacity. Therefore, the average response time is also used. This is demonstrated in Figure 3.5.

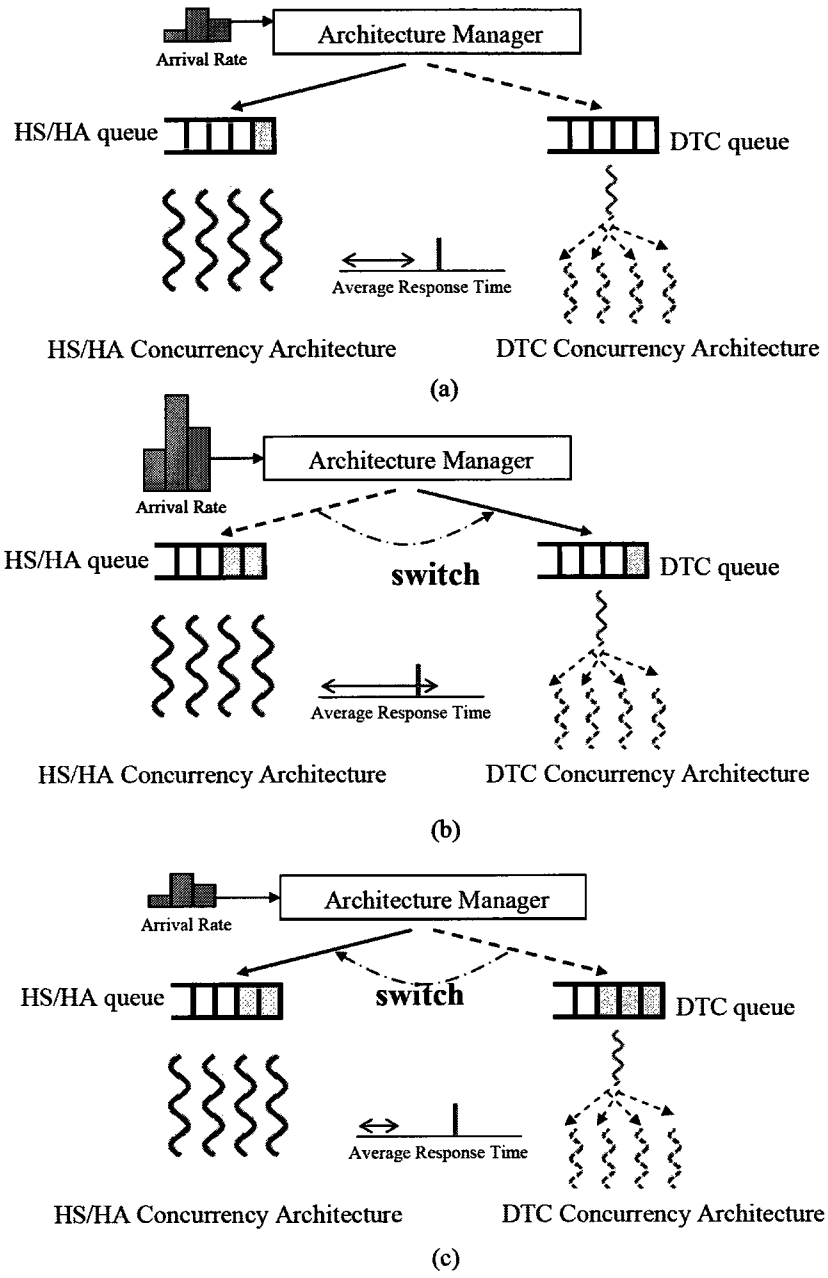

Figure 3.5: Arrival Rate and Response Time-based Self-Adaptive Policy

Initially, when the arrival rate is below a certain level, as shown in Figure 3.5 (a), the *Architecture Manager* passes new requests to one queue. When an arrival burst occurs, as

shown in Figure 3.5 (b), the arrival rate exceeds a threshold and the response time also increases over a certain level. In response, the *Architecture Manager* begins to place new requests to another queue. When the burst is over and the arrival rate returns to the pre-burst level, as shown in Figure 3.5 (c), the *Architecture Manager* puts new requests into the original queue. Details on how to set those thresholds and a simple approach, by monitoring arrival rate and average response time, to detect burst will be discussed in Chapter 4.

3.3.2 Reliability Solution

If the request arrival rate increase or stays the same but the throughput of the system drops to zero, the concurrency architecture is assumed to have failed. In this case, the *Architecture Manager* begins sending requests to another concurrency architecture, as depicted in Figure 3.6 (b). The idea is to demonstrate the applicability of the framework. More sophisticated approaches that can detect failures can be adopted without affecting the framework.

When a system fails, reboot is typically used. Unlike many existing works, Candea, at el [Can04] present the idea of micro-rebooting. Micro-rebooting keeps track of system states and avoids restart from the very beginning. As a result, the system can recover much more quickly when a failure occurs. Similarly, the rebooting from the start approach is not used in SAFCA, thus making the recovery process much faster. Whitehead, et al [Whi09] postulate that failure recovery can be achieved using the concept of micro-rebooting and feature model. If one feature component fails, another set of feature components are reorganized to bypass the failed future, and the system can

continue to function. In addition, using a different set of feature components can avoid the same failure when the same scenario occurs.

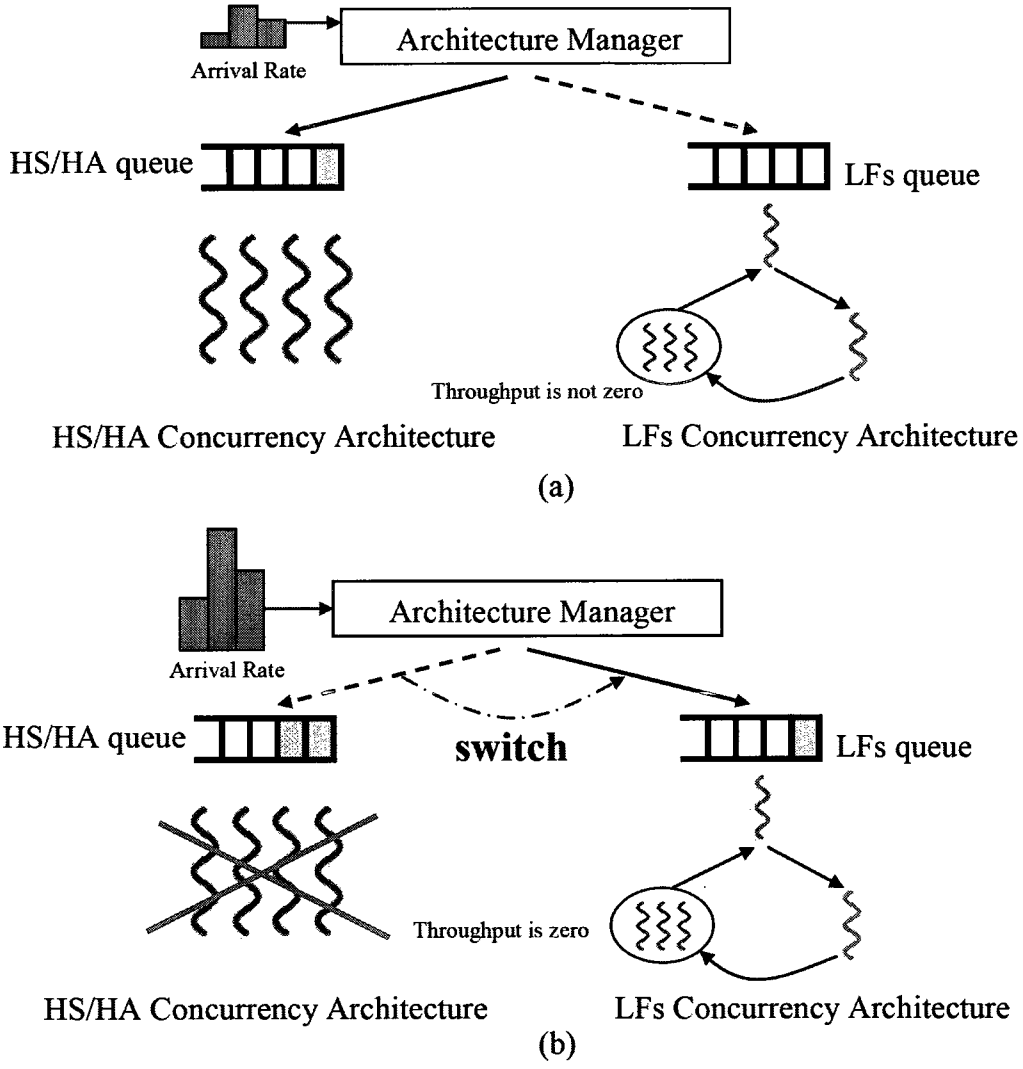

Figure 3.6: Reliability-based Self-Adaptive Policy

The self-healing idea of SAFCA is similar to this approach: if one concurrency architecture fails, requests are put to another concurrency architecture. Each architectural alternative is similar to a feature in this case.

3.4 Architecture Manager

The *Architecture Manager* receives requests through sockets. Depending on the value of a shared object, *Queue Name*, each request is placed in the corresponding queue. The *Executor* component is responsible for updating the value of *Queue Name*.

The *Architecture Manager* is a producer and implemented as an asynchronous thread. It is a producer because it fills a queue with content or requests. It is an asynchronous thread because it responds to I/O requests (requests from the sockets) asynchronously. This thread has a high scheduling priority so it can respond to incoming requests more or less immediately. Once the destination queue becomes full, the *Architecture Manager* drops any new requests.

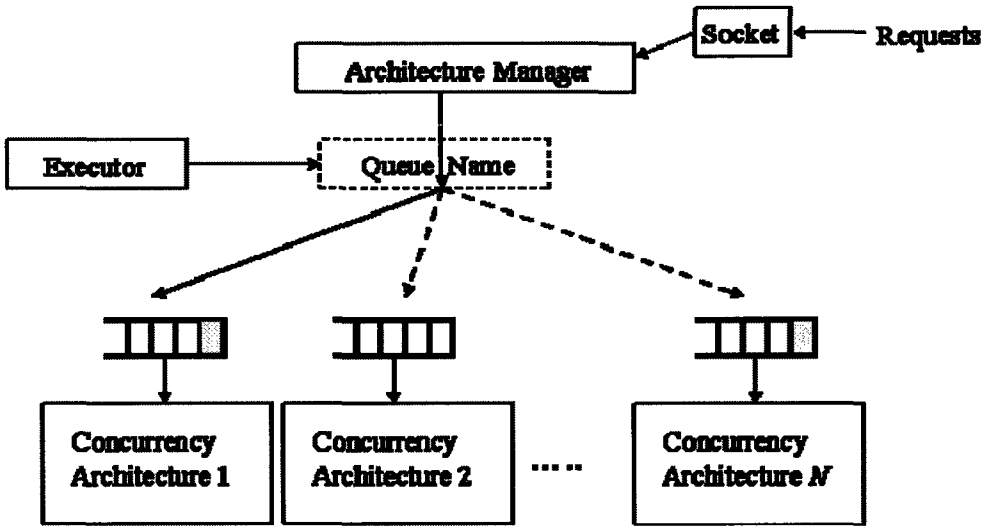

Figure 3.7: High Level Structural View for the Architecture Manager

The *Architecture Manager* contains a linked list for the queues to which it sends requests. Each queue in this list is indexed by its queue name. The value in *Queue Name* corresponds to one of these names.

The shared object *Queue Name* is protected, since two threads (the other one being the *Executor*) access it concurrently. The *Architecture Manager* may have to wait to gain access to *Queue Name*. Since there are only two threads and the *Architecture Manager* has a higher priority, the wait time can be ignored.

The *Architecture Manager* also performs other functions that are essential to the *Monitor* component. The *Architecture Manager* records the rate of request arrival and the number of requests dropped. This information is essential for the *Monitor* component. Additionally, for each newly arrived request, the *Architecture Manager* adds a timestamp to the request. This timestamp will be used by the *Monitor* component to calculate the response time for the request.

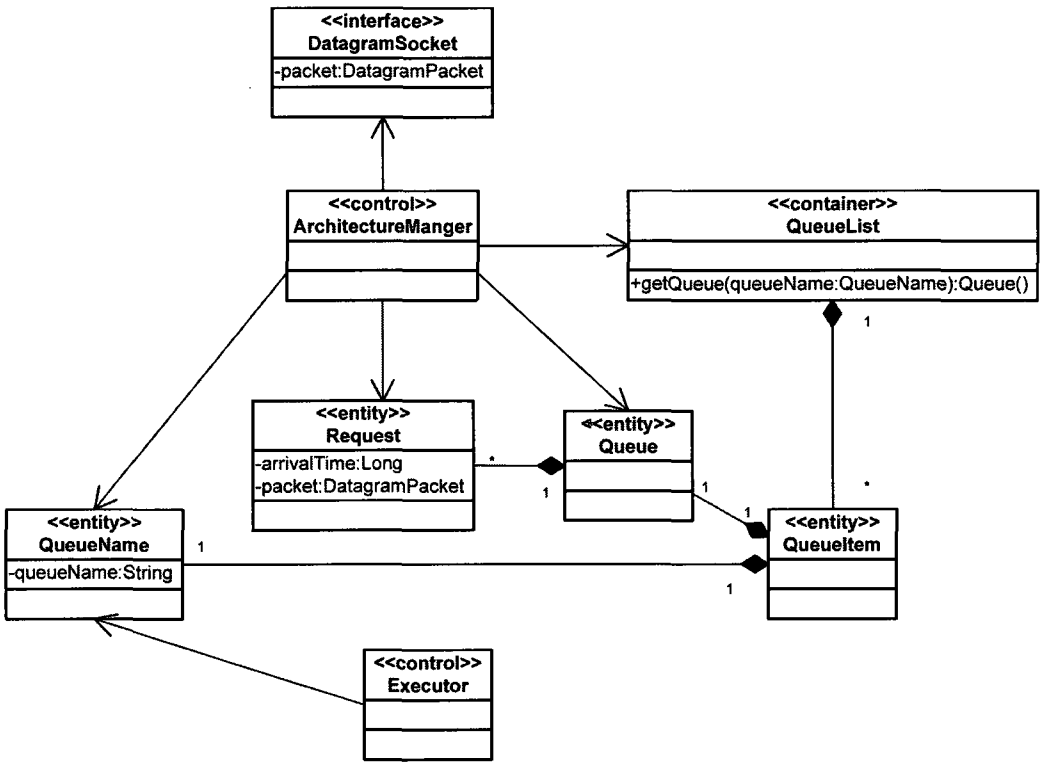

Figure 3.8: Class Diagram for the Architecture Manager

Figure 3.8 presents the class diagram of the *Architecture Manager*. The *Architecture Manager* class is associated with several classes: *DatagramSocket* class, *Request* class, *QueueList* class, *QueueName* class, and *Queue* class. Java datagram socket supports UDP packets and UDP is a connection-less protocol. The *QueueList* contains several *QueueItems*. Each *QueueItem* contains a *QueueName* and a *Queue*. Each *QueueName* is bound to a *Queue*, serving as an index. Each *Queue* is serviced by a different concurrency architecture. To introduce a new concurrency architecture into the framework, one only has to add a new *Queue* and the corresponding *QueueName* so the new concurrency architecture can receive requests from that *Queue*.

The dynamics of the *Architecture Manager* are demonstrated in Figure 3.9. When a new request arrives, the *Architecture Manager* receives it from the *DatagramSocket* and then creates a *Request* object. This object contains the received data packet and a timestamp is added. This timestamp tracks when the request was received and will be used to calculate the response time.

The *Architecture Manager* reads the shared object *QueueName*. The value is used to indicate which *Queue* to store the *Request*. The *Architecture Manager* then checks the queue size. If the *Queue* is full, the *Request* is dropped. If the *Queue* is not full, the *Request* is added to the *Queue*. The shared object *QueueName* is also accessed by the *Executor*. The *Executor* changes the value of *QueueName* to notify the *Architecture Manager* the appropriate queue to send *Request* so that the system can adapt to the appropriate concurrency architecture to serve the *Request*. A synchronization mechanism is used to protect the shared object *QueueName* to ensure mutual exclusion.

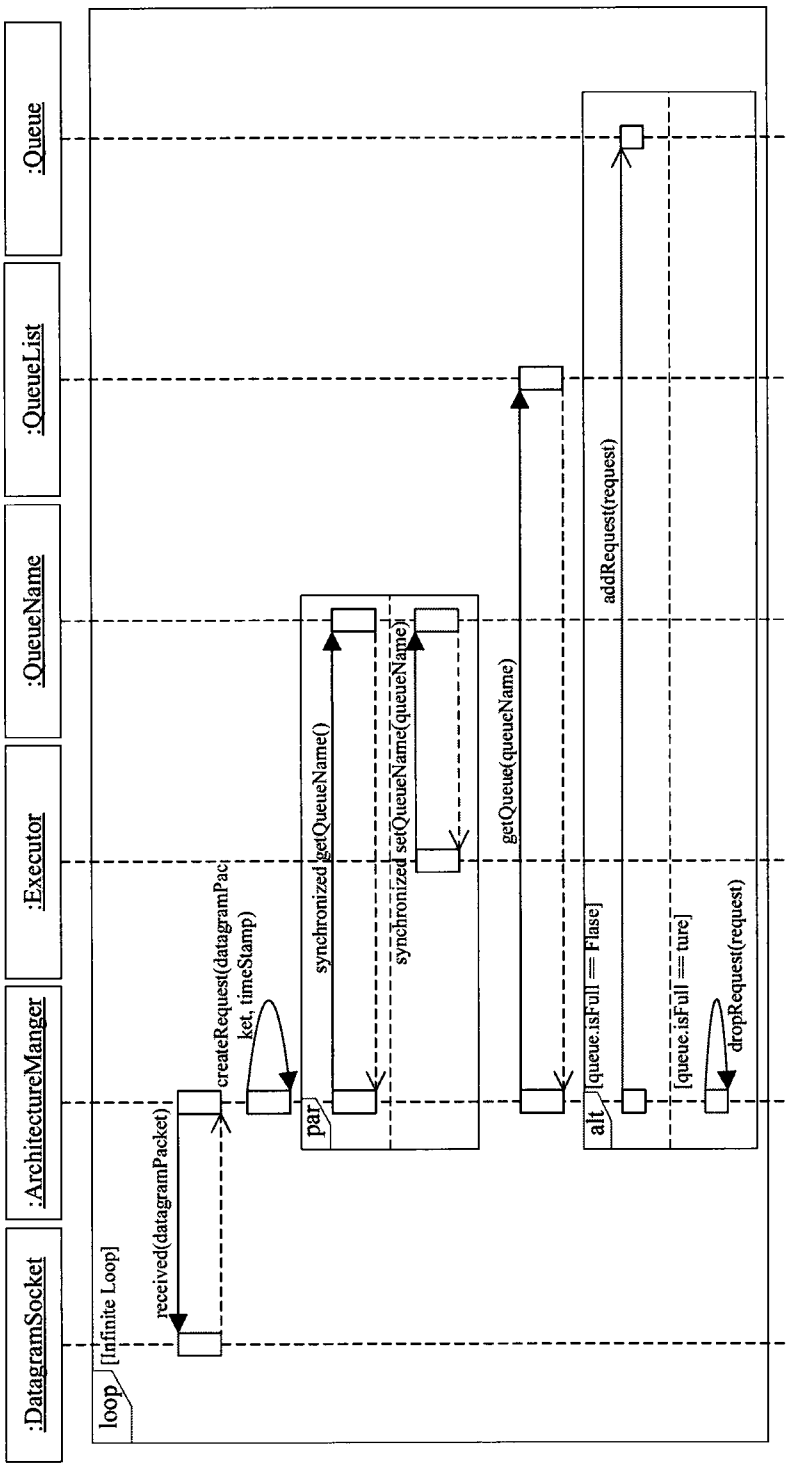

Figure 3.9: Sequence Diagram for the Architecture Manager

3.5 Design of Concurrency Architectures

SAFCA is designed to be scalable and flexible by using an object-oriented design pattern called the Factory design pattern [Mar99]. Therefore new concurrency architectures can be easily added into the framework. The Factory design pattern is classified as a creational pattern. The goal of creational patterns is to create objects without specifying the class type for the object. The advantage of this approach is that one can easily introduce a new object type into the program.

The Factory design pattern achieves the ease of introducing a new object type as follows: the Factory (a class) creates products (objects) through a specific Factory method. The subclasses of the product class then override the product class, thus specifying the derived type of product that will be created. To demonstrate the Factory design pattern, a simple example is used. With three different types of sandwiches, each with a different price, we would like to create various classes to represent the different types of sandwiches.

Figure 3.10 shows that the subclasses of the *SandwichFactory* override the *createSandwich*() method to create the specific type of sandwich. Additionally, the subclasses inherit the *selectBread*() method, so each subclass does not have to implement this method again. Each subclass of *Sandwich* also overrides the *getPrice*() method to return the correct price for the specific sandwich. The *getBread*() method and *theBread* attribute, on the other hand, are inherited by the sandwich subclasses.

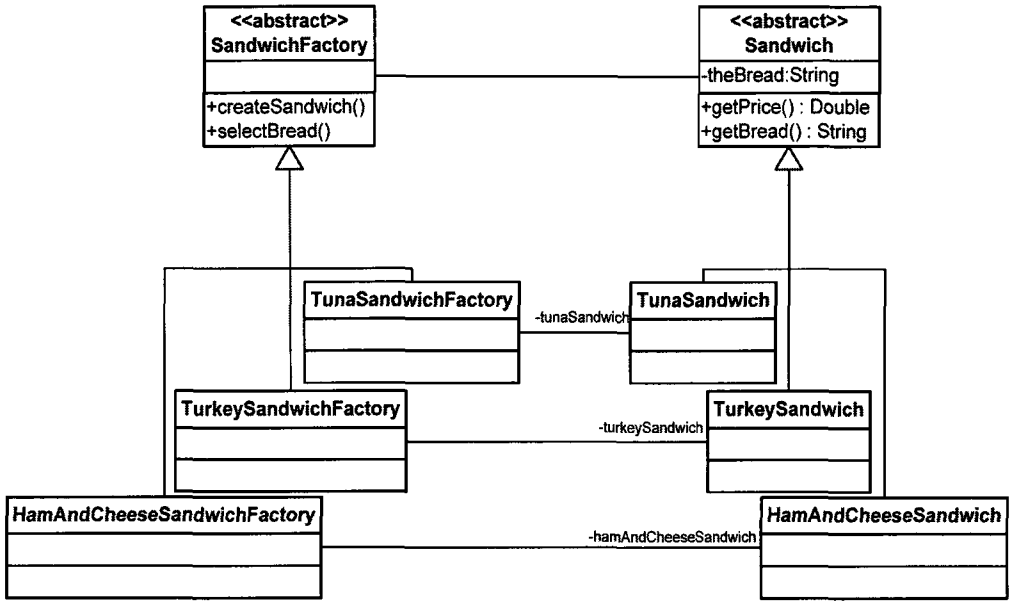

Figure 3.10: Class Diagram for the Sandwich Factory

Since every concurrency architecture needs to create worker threads to process requests, the worker threads are the products in our design, while the concurrency architectures are the factories in the Factory design pattern context. Similar to the subclasses of *Sandwich*, the different worker threads corresponding to different architectures realize the same functions by inheriting the methods from the parent class.

As mentioned in Section 3.4, if a new concurrency architecture needs to be added, it should access the appropriate queue that receives the request. It also needs to create the appropriate worker threads to process requests. Those functions have already been defined. By using the Factory design pattern, a new concurrency architecture does not have to define those functions again after it extends the abstract class. Therefore, a new concurrency architecture can be easily added into the framework.

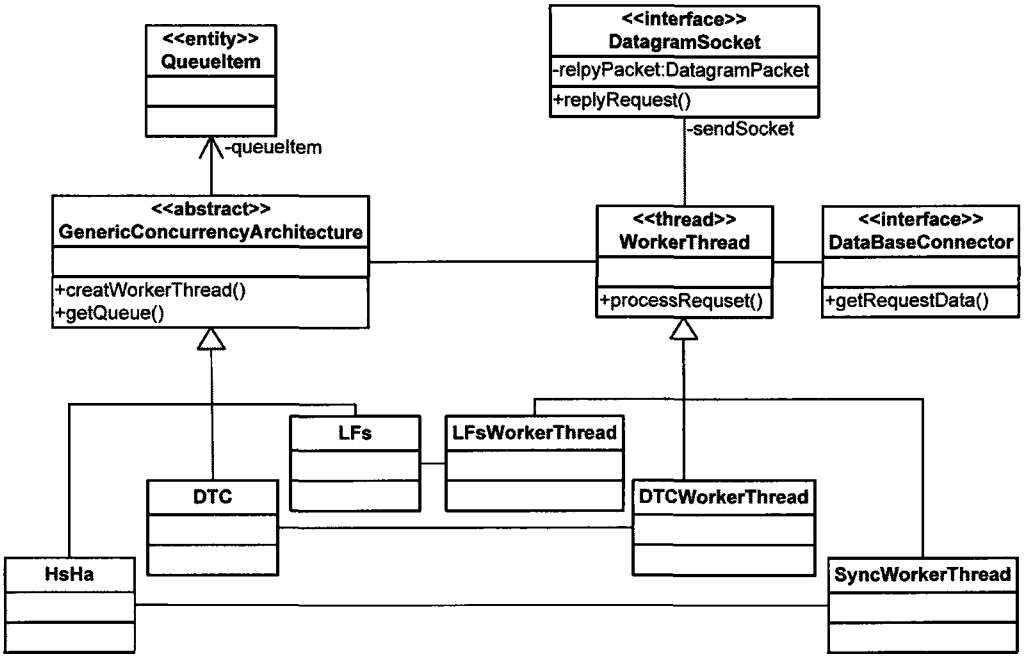

Figure 3.11: Class Diagram for SAFCA

As demonstrated in Figure 3.11, the *LFs*, *DTC*, and *HsHa* subclasses represent three concurrency architectures. These three subclasses extend the abstract class *GenericConcurrencyArchitecture* that defines an abstract method *CreateWorkerThread*() that is overridden by the subclasses. The *CreateWorkerThread*() in each subclass creates the *workerThread* instance corresponding to the architecture represented by that subclass. For example, *HsHa* creates *SyncWorkerThread* instances. The *getQueue*() method in the abstract class is inherited by the subclasses to get requests from the queue corresponding to that subclass. In the self-adaptive framework, *LFsWorkerThread*, *DTCWorkerThread*, and *HsHaWorkerThread* are all subclasses of *WorkerThread*. *WorkerThread* has two interfaces: *DatabaseConnector* and *DatagramSocket*. The method *getRequest*() in

DatabaseConnector is used to communicate with database while the method *replyRequest()* in *DatagramSocket* is used to send reply messages to clients. These two methods are I/O-bound operations. The method *processRequest*() in *WorkterThread* represents all CPU-bound operations required to process a request. These three methods are inherited all by the subclasses of *WorkerThread*.

The following subsections explain how the worker threads in different concurrency architectures are created and how those worker threads access their queue.

3.5.1 Design of Half-Sync/Half-Async

As mentioned in Section 2.3.5, HS/HA concurrency architecture consists of three layers: the asynchronous layer, the queuing layer, and the synchronous layer. The asynchronous layer receives requests from sockets asynchronously and places them in the queuing layer. A number of threads in the synchronous layer read the requests from the queuing layer and process them synchronously. Figure 3.12 demonstrates that the queue is accessed by both the *Architecture Manager* (acting as the asynchronous thread) and the *SyncWorkerThreads*.

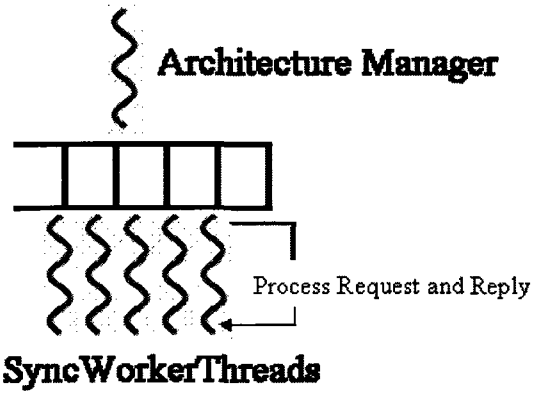

Figure 3.12: Half-Sync/Half-Async Structure

Once the *Architecture Manager* begins to send requests to the HS/HA queue, since the *Architecture Manager* is implemented as a thread, it becomes the asynchronous thread in the HS/HA concurrency architecture. The threads in the thread pool become the synchronous threads in the HS/HA concurrency architecture.

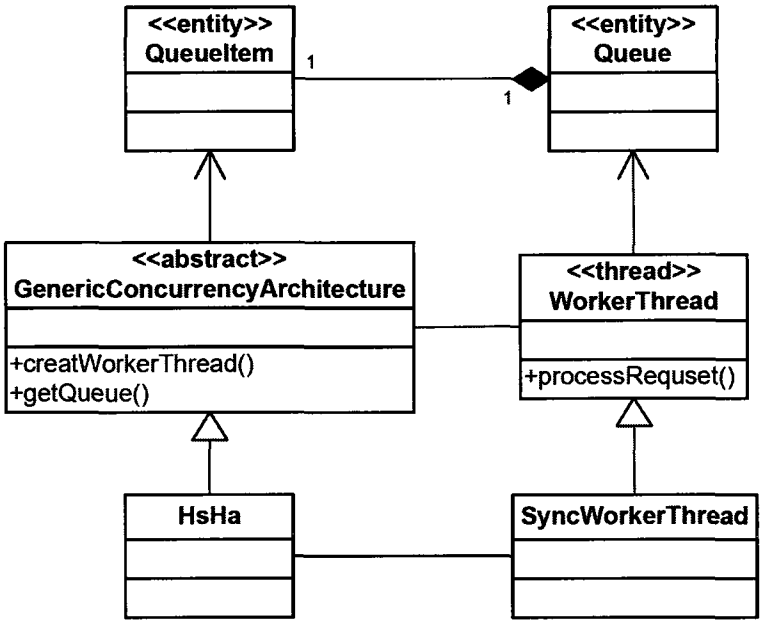

Figure 3.13: Class Diagram for the HS/HA Concurrency Architecture

As depicted in Figure 3.13, the *HsHa* subclass represents the HS/HA concurrency architecture and extends the abstract *GenericConcurrencyArchitecture* class. The *SyncWorkerThread* extends the *WorkerThread* class. The *HsHa* class has a reference to the queue it should get requests from and the method *getQueue()* returns this reference. The *HsHa* object creates worker threads (*SyncWorkerThread)*. Each worker thread invokes the *getQueue()* method to get a reference to the appropriate queue, removes a request from the queue, and processes the request.

Figure 3.14 demonstrates the sequence diagram for the HS/HA concurrency architecture. The *HSHA* object creates a number of worker threads by invoking *createWorkThread()* repeatedly.

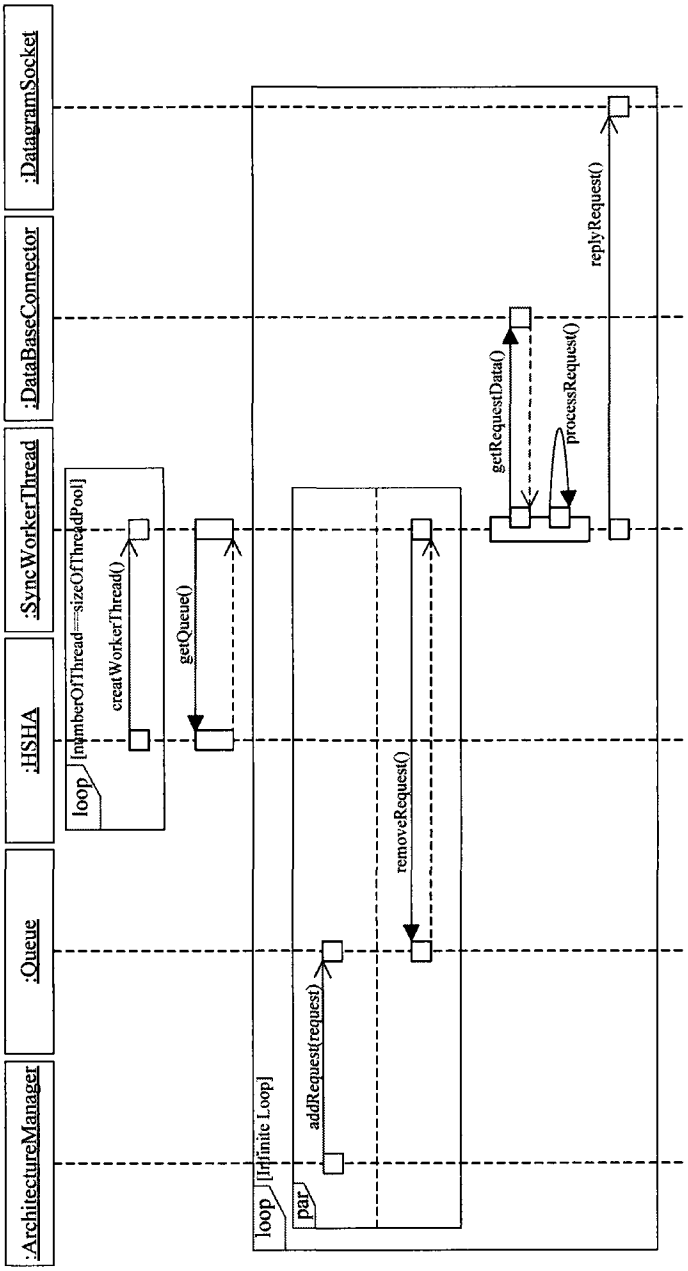

Figure 3.14: Sequence Diagram for the HS/HA Concurrency Architecture

The number of work threads created is the size of the thread pool. Once all the threads are created, all threads start. Each thread first invokes *getQueue()* to obtain a reference to the appropriate queue. The thread then goes into an infinity loop: If the queue is empty, the thread goes into wait state. If the queue is not empty, the thread removes a request from the queue (the access to the queue is protected by synchronization), gets the request related data, processes the request, and sends a reply for this request, after which the loop starts again.

3.5.2 Design of Dynamic-Thread-Creation

DTC creates a thread for each newly arrived request, and each thread self-terminates after it has finished processing a request, as shown in Figure 3.15.

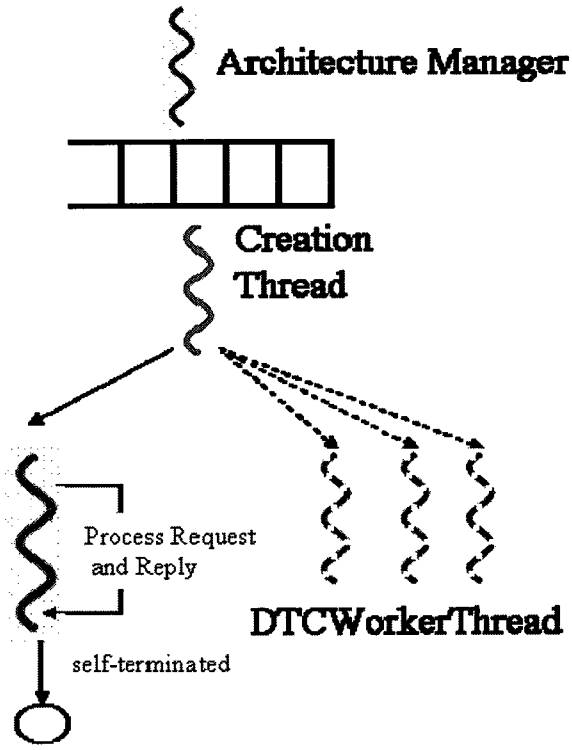

Figure 3.15: Dynamic-Thread-Creation Structure

The *Architecture Manager* puts requests in the queue, while the *DTC* creation thread removes the request from the queue. The creation thread then creates a new *DTCWorkerThread* to process this request.

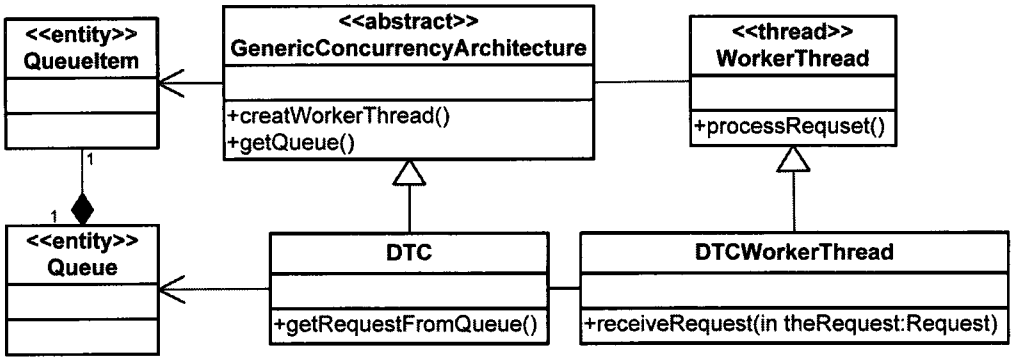

Figure 3.16: Class Diagram for the DTC Concurrency Architecture

The architecture of the DTC is illustrated in Figure 3.16. The *DTC* subclass represents the DTC concurrency architecture and extends the abstract *GenericConcurrencyArchitecture* class. The *DTCWorkerThread* extends the *WorkerThread* class. Similar to *HsHa* described in Section 3.5.1, *DTC* also has a reference to the queue it should get requests from and the method *getQueue()* returns this reference. However, the implementation of the *DTC* subclass differs from the *HsHa* subclass. *DTC* implements the Java *Runnable* interface and thus can run as a thread. This thread acts as the creation thread. It removes a request from the queue, creates a *DTCWorkerThread*, and passes the request to the worker thread. As a result, the worker threads do not have to access the queue. This is different from the *SyncWorkerThread* in *HsHa* where each worker thread has to access the queue. The *getRequestFromQueue*() method in *DTC* invokes the *getQueue()* method to determine the correct queue to receive

requests and get a request. The request is given to the newly created *DTCWorkerThread* by invoking the *receiveRequest()* method.

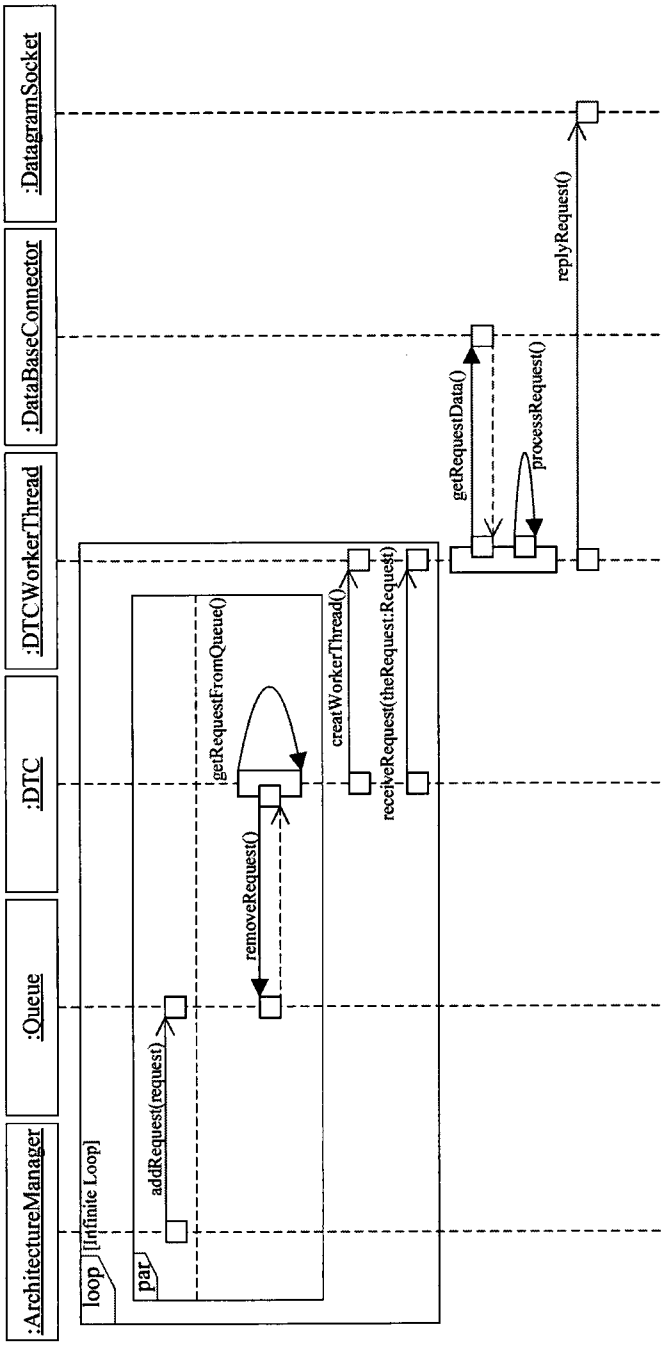

Figure 3.17: Sequence Diagram for the DTC Concurrency Architecture

Figure 3.17 depicts the sequence diagram of the DTC architecture. *DTC* repeats the following steps: First, it checks if the queue is empty. If the queue is empty, the *DTC* goes into the wait state. If the queue is not empty, *DTC* removes a request from the queue (the access to the queue is protected by synchronization), creates a new worker thread, and assigns the request to the worker thread.

After a worker thread receives a request, it processes the request, sends out a reply, and dies, following which the Java garbage collector frees any resources used by this thread.

3.5.3 Design of Leader/Followers

Figure 3.18 shows a LFs thread has three possible states: leader state, processing state, and follower state.

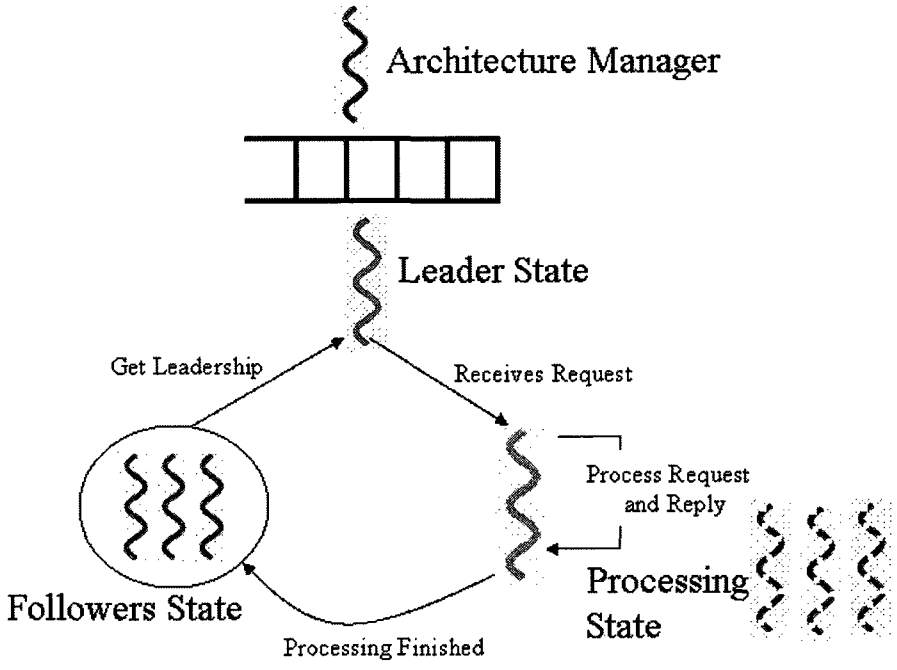

Figure 3.18: Leader/Followers Structure

In the leader state, the LFs thread receives a request from the queue and immediately promotes another thread to the leader state. In the processing state, the LFs thread processes a request. When the processing is finished, the LFs thread transits from the processing state to the follower state. In the follower state, the thread waits to be promoted to the leader state. There is only one thread in the leader state at one time.

As shown in Figure 3.16, the *LFs* subclass represents the LFs concurrency architecture and extends the abstract *GenericConcurrencyArchitecture* class. The *LFsWorkerThread* extends the *WorkerThread* class. The *leaderShip* attribute in *LFs* is used to change the state of the *LFsWorkerThread*. The *LFsWorkerThread* requests to be changed from the follower state to the leader state by invoking the *getLeaderShip*() method in *LFs* class. The *promoteNewLeader*() method in *LFs* class is used to prompt a *LFsWorkerThread* in the follower state to the leader state.

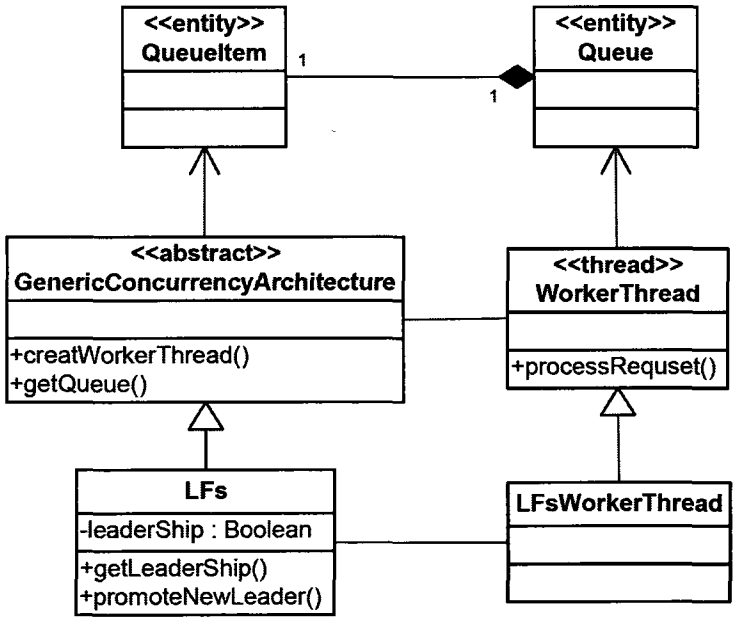

Figure 3.19: Class Diagram for the LFs Concurrency Architecture

Figure 3.20 presents the sequence diagram for the LFs concurrency architecture. The *LFs* first creates a number of *LFsWorkerThreads* by repeatedly invoking *createWorkerThread*().

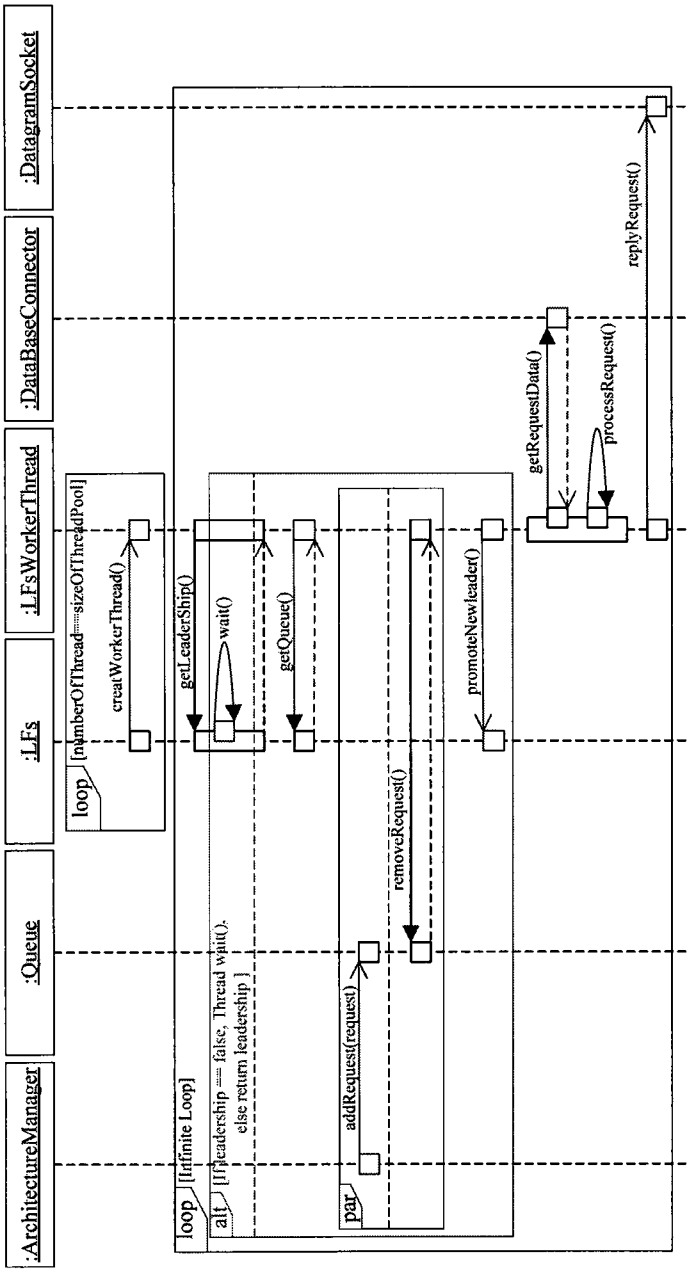

Figure 3.20: Sequence Diagram for the LFs Concurrency Architecture

Each *LFsWorkerThread* iteratively tries to become the leader by calling *getLeaderShip*(). If the leadership has been taken, it will wait until it is promoted to leader. It then removes a request from the queue and immediately prompts another follower to leader. The request is then processed and a reply is sent out.

3.6 Design of Decider

The *Decider* component has several functions: monitoring, analyzing, and deciding. As stated in Section 3.3, the design of the *Decider* depends on one of two types of self-adaptive policies: Queue Length-based Self-Adaptive Policy or Arrival Rate and Response Time-based Self-Adaptive Policy. These policies are based on threshold control, since multi-tiered systems threshold control can offer a straightforward way to adaptively adjust capacity provisioning in terms of response time or to improve performance for overload systems [Urg05][Wel03].

3.6.1 Queue Length-based Self-Adaptive Policy

In this self-adaptive policy, the *Decider* polls the length of every concurrency architecture queue at regular sampling intervals. If one of the queues reaches its limit, the *Decider* notifies the *Executor* to send subsequent requests to a different concurrency architecture queue. The overloaded queue is still serviced by its concurrency architecture until the queue length is reduced below a certain threshold The *Decider* then notifies the *Executor* to divert requests back to this queue. The value of this threshold impacts the performance of the overall system. How to set this threshold is studied in Chapter 4. SAFCA implemented using this self-adaptive policy is denoted as SAFCA-Q.

3.6.2 Arrival Rate and Response Time-based Self-Adaptive Policy

The *Decider* in this self-adaptive policy requires some information provided by the *Architecture Manager* and by the individual concurrency architecture.

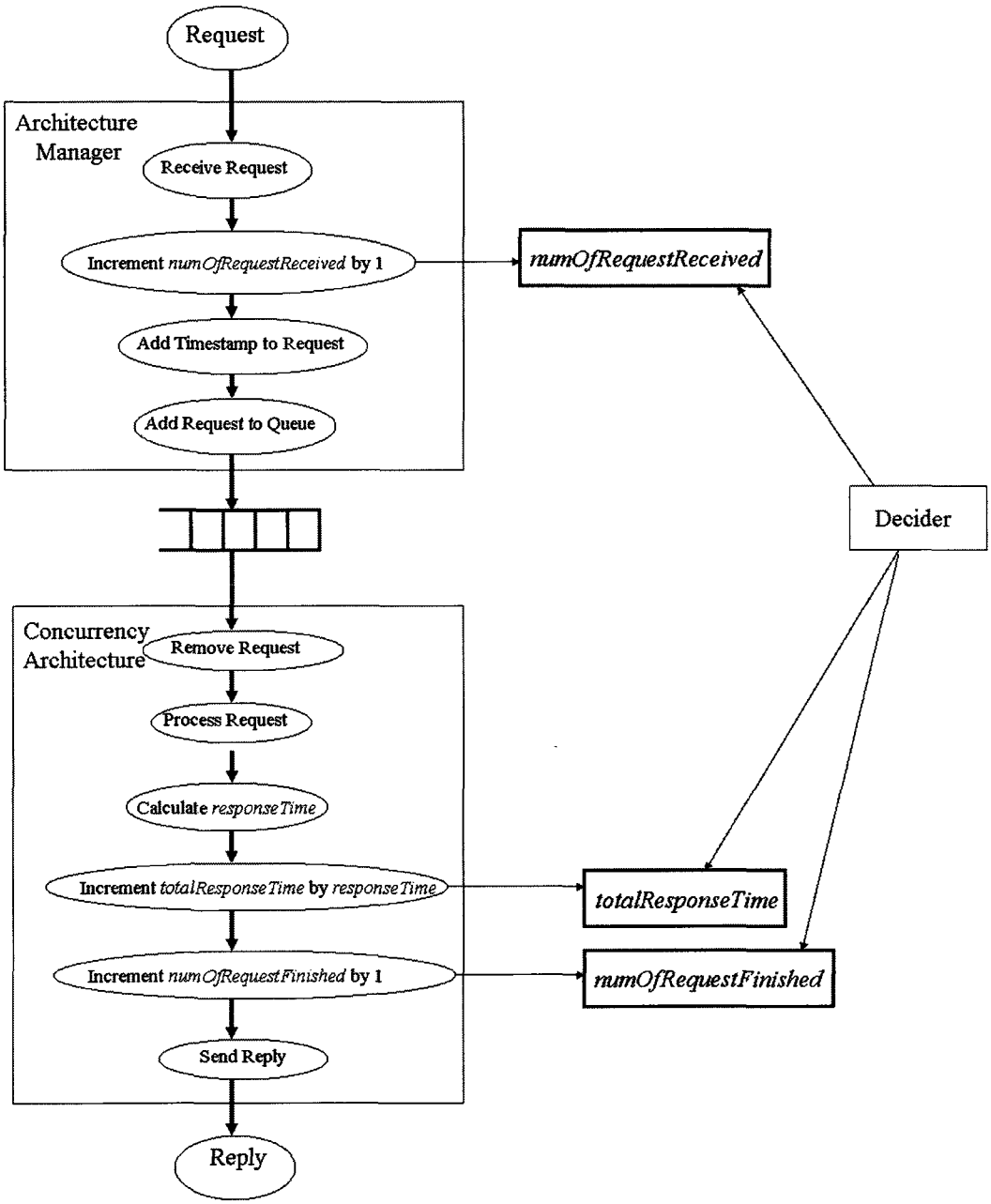

Figure 3.21: Arrival Rate and Response Time-based Self-Adaptive Policy

When a request arrives at the *Architecture Manager*, the *Architecture Manager* increments the *numOfRequestReceived* counter by one. A timestamp *ArrivalTime* is added to the request message and the request message is placed to the queue.

After the concurrency architecture removes a request from the queue, the request is processed. The *responseTime* for this request is calculated using the *FinishTime* and the *ArrivalTime*. The *totalResponseTime* is incremented by the *ResponseTime* and the *numOfRequestFinished* is increased by one. Finally, a reply is sent to the client. The values of *numOfRequestReceived*, *numOfRequestFinished*, and *totalResponseTime* are polled by the *Decider* at the end of every sampling interval. Those values are set to zero at the beginning of every interval.

The response time for the *i*-th a request in a sampling interval is calculated as equation (1):

$$\text{ResponseTime}_{(i)} = \text{FinishTime}_{(i)} - \text{ArrivalTime}_{(i)} \tag{1}$$

$$\text{totalResponseTime} = \sum_{i=1}^{\text{numOfRequestFinished}} \text{ResponseTime}_{(i)} \tag{2}$$

Decider uses the values in *numOfRequestReceived*, *totalResponseTime*, and *numOfRequestFinished* to calculate *avgArrivalRate* (average arrival rate), *avgResponseTime* (average response time) in a sampling interval, with a length of *samplingIntervalLength*, by using equation (3) and equation (4):

$$\text{avgArrivalRate} = \frac{\text{numOfRequestReceived}}{\text{samplingIntervalLength}} \tag{3}$$

$$\text{avgResponseTime} = \frac{\text{totalResponseTime}}{\text{numOfRequestFinished}} \quad (4)$$

Decider calculates the *avgArrivalRate* and the *avgResponseTime* in each interval. There are two approaches to using those two values to decide whether an action should be taken. When a sudden burst of requests arrives, while the *avgArrivalRate* will have a sudden increase, it does not imply that the system is overloaded and actions are required. If the *avgResponseTime* also experiences a significant increase, then the *Decider* will decide to take action.

Burst detection is a very complex research topic. This work only provides two basic approaches for burst detection. While more sophisticated burst detection approaches can be incorporated into our framework, they are beyond the scope of this work. Our approaches are based on thresholds and historical data. Those two policies are explained further next.

Arrival Rate and Response Time-based Self-Adaptive Policy A

As depicted in Figure 3.22, the solid line represents the response time for the self-adaptive system while the dotted line is the response time for the non-adaptive system. In our experiments, there is a pre-configured normal arrival rate (*normalArrivalRate*) for normal conditions and a burst arrival rate (*burstArrivalRate*) when a burst occurs. The sampled arrival rate of the previous interval is *prevArrivalRate* and the sampled arrival rate of the current interval is *curArrivalRate*. Similarly, *preResponseTime* is the response time of the previous interval, and *curResponseTime* is the response time of the current interval.

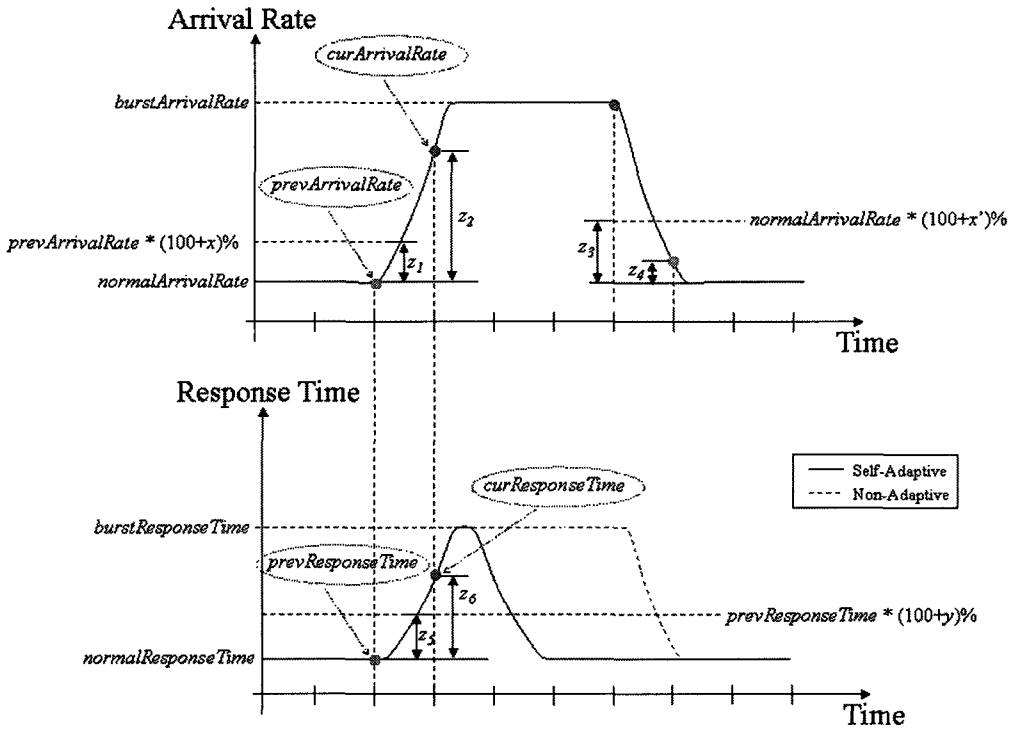

Figure 3.22: Burst Detection Policy A

Figure 3.22 demonstrates burst detection policy A. Six parameters are involved in the policy. z_1 represents how much increase in the arrival rate is needed for burst detection. z_2 is the amount of increase in the arrival rate. z_3 is the amount of decrease in the arrival rate that is used to detect if the burst is finished. z_4 is the amount of decrease in the arrival rate. z_5 represents how much increase in the response time is needed to detect system overload. z_6 is the amount of increase in the response time. The following equations are used to calculate z_1 to z_6 in Figure 3.22.

$$z_1 = \text{prevArrivalRate} * x\% \quad (5)$$

$$z_2 = \text{curArrivalRate} - \text{prevArrivalRate} \quad (6)$$

$$z_3 = \text{normalArrivalRate} * x'\% \quad (7)$$

$$z_4 = \text{curArrivalRate} - \text{prevArrivalRate} \quad (8)$$

$$z_5 = \text{prevResponseTime} * y\% \quad (9)$$

$$z_6 = \text{curResponseTime} - \text{prevResponseTime} \quad (10)$$

As shown in the burst detection pseudo code in Figure 3.23, if the difference (z_2) between the current arrival rate (*curArrivalRate*) and previous arrival rate (*prevArrivalRate*) is greater than x% of the previous arrival rate (*preArrivalRate*), then a burst is assumed to have occurred. However, this does not necessarily mean that the system is overloaded and self-adaptive action needs to be taken. If the difference (z_6) between the current response time (*curResponseTime*) and previous response time (*preResponseTime*) also jumps by y% of the previous response time (*preResponseTime*), then the *Decider* notifies the *Executor* to take actions and store the value of *preArrivalRate* to *normalArrivalRate*.

```
BEGIN
   IF z2 > z1  and z6 > z5
     normalArrivalRate = prevArrivalRate
     The Decider notify the Executor to take adaptive action (allocate more resources)
   ENDIF
   IF z4<z3 and curArrivalRate < prevArrivalRate
    The Decider notify the Executor to take adaptive action (free resources)
   ENDIF
END
```

Figure 3.23: Pseudo Code for Burst Detection Approach A

If the current arrival rate (*curArrivalRate*) is less than the previous arrival rate (*prevArrivalRate*) and their difference (z_4) is greater than x'% of the previous arrival rate

(*preArrivalRate*), then the burst is assumed to be over. The *Decider* notifies the *Executor* to free resources.

Figure 3.22 demonstrates that the response time of a non-adaptive system rises and falls following the burst, while our self-adaptive approach reduces the response time. Further analysis is provided in Chapter 4. Chapter 4 will also discuss how to choose the value for x, x' and y through experimentation and analysis. SAFCA designed using burst detection approach A is denoted as SAFCA-A.

Arrival Rate and Response Time-based Self-Adaptive Policy B

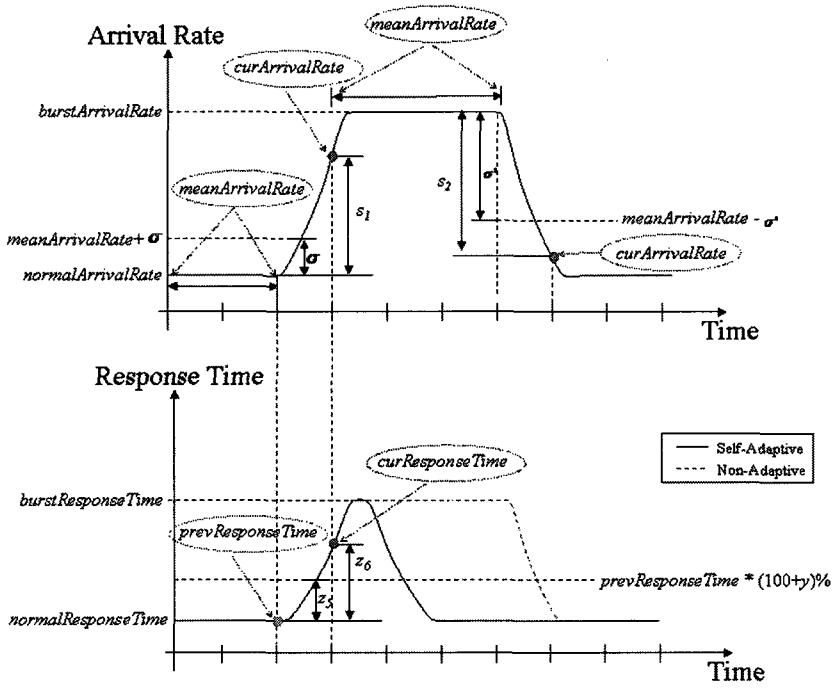

Figure 3.24: Burst Detection Policy B

As depicted in Figure 3.24, the solid line represents the response time for the self-adaptive system while the red-dotted line is the response time for the non-adaptive

system. Burst detection policy B differs from policy A in two aspects: 1) instead of using the arrival rate of the previous interval (*prevArrivalRate*), the mean rate of all the arrivals so far (*meanArrivalRate*) is used. 2) Instead of using a percentage, the standard deviation (σ) of the previous arrival rates is used. The idea is presented in Figure 3.24.

Those quantities in Figure 3.24 are calculated as follows:

Assume $r_1, r_2, r_3 \ldots r_n$ are arrival rates for sampling interval 1, 2, 3 .... n.

$$\text{meanArrivalRate} = \frac{\sum_{i=1}^{n} r_i}{n} \tag{11}$$

$$\sigma = \sqrt{\frac{\sum_{i=1}^{n} (r_i - \text{meanArrivalRate})^2}{n}} \tag{12}$$

$$s_1 = \text{curArrivalRate} - \text{meanArrivalRate} \tag{13}$$

$$s_2 = \text{curArrivalRate} - \text{meanArrivalRate} \tag{14}$$

As depicted in the burst detection pseudo code in Figure 3.25, if the difference (s_1) between the current arrival rate (*curArrivalRate*) and the mean arrival rate (*meanArrivalRate*) is greater than the standard deviation of previous arrival rates (σ), then a burst is assumed to have occurred. However, this does not mean that the system is overloaded and self-adaptive action needs to be taken. If the difference (z_6) between the current response time (*curResponseTime*) and previous response time (*preResponseTime*) also increases by *y*% of the previous response time (*preResponseTime*), then the *Decider* notifies the *Executor* to take actions and *meanArrivalRate* is reset to zero. From this point on, a new *meanArrivalRate* and a new standard deviation will be calculated.

```
BEGIN
   IF s1 > σ  and z6 > z5
     The Decider notify the Executor to take adaptive action (allocate more resources)
      meanArrivalRate = 0
      σ =0
   ENDIF
   IF s2 < σ
     The Decider notify the Executor to take adaptive action (free resources)
      meanArrivalRate = 0
      σ =0
   ENDIF
END
```

Figure 3.25: Pseudo Code for Burst Detection Approach B

If the current arrival rate (*curArrivalRate*) is less than the mean arrival rate (*meanArrivalRate*), and their difference (s_2) is greater than the standard deviation of the arrival rates (σ), then the burst is assumed to be over. The *Decider* notifies the *Executor* to free resources and *meanArrivalRate* is reset to zero. From this point on, a new *meanArrivalRate* and a new standard deviation will be calculated. SAFCA implemented using burst detection approach B is denoted as SAFCA-B.

3.6.3 Reliability-based Self-Adaptive Policy

The self-adaptive framework can also be used to improve reliability in case of a failure. If the current arrival rate (*curArrivalRate*) increases or stays the same from the previous interval, but the number of requests processed (*numOfRequestsFinished*) becomes zero, the *Decider* decides that the concurrency architecture has failed. The *Decider* then notifies the *Executor* to change *queueName* to anther concurrency architecture. Currently, two concurrency architectures, HS/HA and LFs, have been adopted in our experiments to support this policy. If the *Decider* determines that the

HS/HA architecture fails, the LFs architecture will take over the processing of requests. SAFCA designed to improve reliability is denoted as SAFCA-R. Section 4.9 will demonstrate this policy further.

3.7 Design of Executor

The main purpose of Executor is to instruct the *Architecture Manager* to put new requests into the appropriate concurrency architecture queue. The *Executor* interacts with the *Decider* and modifies the shared object *QueueName* to control which queue the *Architecture Manager* sends request to.

Figure 3.26 illustrates the class diagram of the *Decider* and the *Executor*. The *Decider* notifies the *Executor* to take an adaptive action through the *AdaptiveSignal* shared object. The value of the attribute *architectureName* of *AdaptiveSignal* is used to set the value of the shared object *QueueName* (*QueueName* has been explained in Section 3.4).

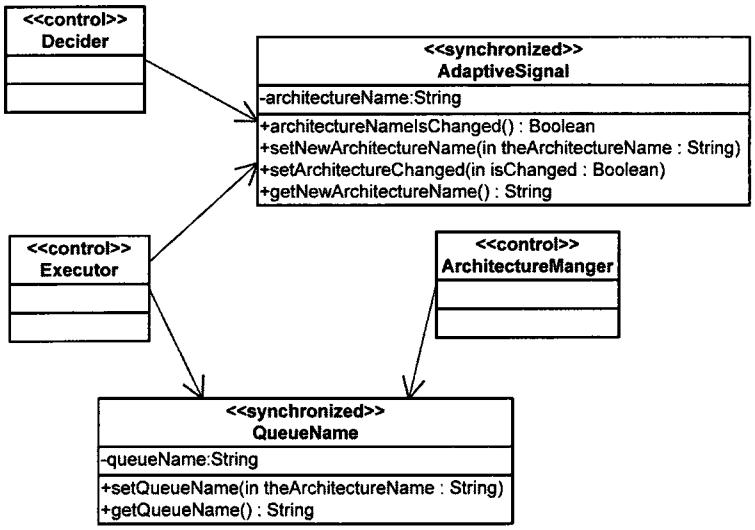

Figure 3.26: Class Diagram for the Executor and the Decider

The *Executor* checks whether *architecutreName* has been changed by invoking the *architectureNameChanged*() method. On the other hand, the *Decider* invokes the *setArchitectureNameChanged*() method to indicate to the *Executor* that there has been a change. The value of *architectureName* is set by the *Decider* by using the *setArchitectureName*() method, while the value of *architectureName* is read by the *Executor* by using the *getArchitectureName*() method.

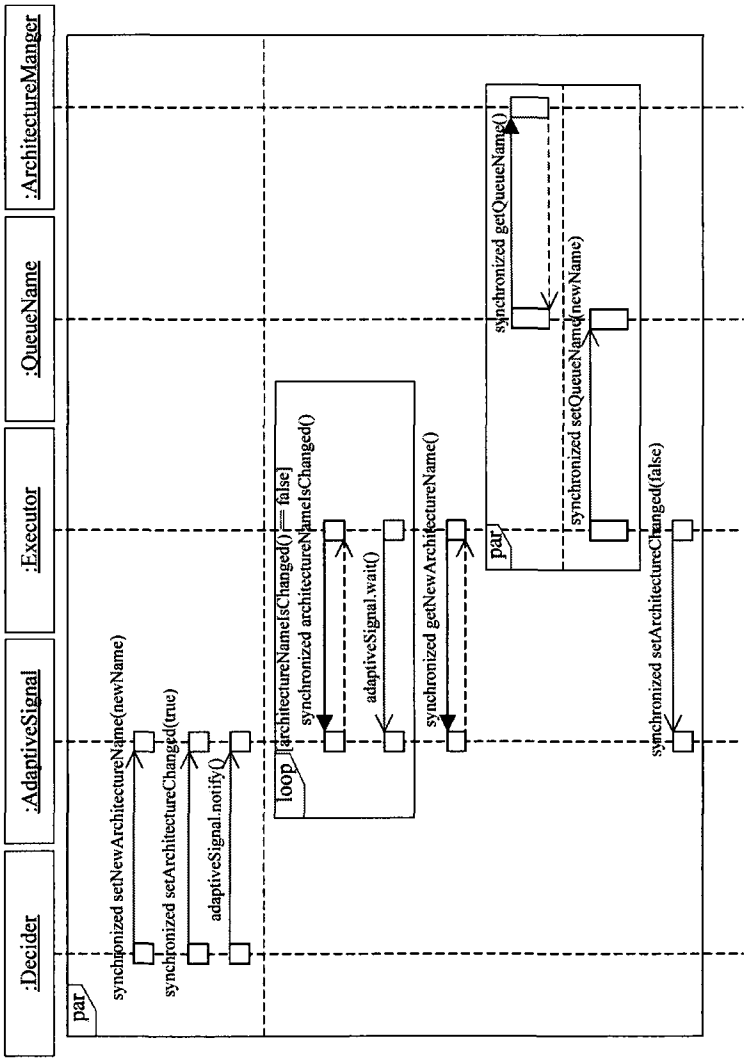

Figure 3.27: Sequence Diagram for the Executor and the Decider

The sequence diagram for the Executor and the Decider is presented in Figure 3.27. The interaction between the *Executor* and the *Decider* is designed based on Java threads and Java thread signalling via shared objects, *wait*(), and *notify*() [Jen08]. Both the *Executor* and the *Decider* are synchronized with the *AdaptiveSignal* shared object. The *Executor* calls *architectureNameIsChanged*(). If the return value is *false*, it goes into wait state until the *Decider* invokes *notify*(). Using the thread signalling via shared object approach, the *Executor* does not have to continuously poll the *architectureName*.

3.8 Summary

This chapter has described the various components in the self-adaptive framework: the *ArchitectureManager*, the *ConcurrencyArchitectures*, the *Decider*, and the *Executor*. Those sections also explained how those components are integrated to form the final framework.

As stated previously, the proposed self-adaptive framework can include many types of concurrency architectures. To handle bursty workloads, the self-adaptive framework in this work focuses on two concurrency architectures: HS/HA and DTC. To improve reliability, the proposed self-adaptive framework involves the HS/HA concurrency architecture and the LFs concurrency architecture.

As explained in Sections 2.3.2 and 2.3.5, HS/HA concurrency architecture (based on the thread pool design) works well under normal workload conditions. However, if a burst occurs, the size of the thread pool becomes the performance bottleneck for HS/HA. On the other hand, as stated in Section 2.3.1, the DTC concurrency architecture can

create a large number of threads to handle the sudden burst of requests (one thread per request). However, the overhead of thread creation and destruction in DTC makes it inferior, compared to HS/HA, under normal workload condition. For this reason, the self-adaptive framework contains both HS/HA and DTC. Normally, HS/HA processes requests from the HS/HA queue. If a burst arrives, requests will be put to the DTC queue instead, so more resources (in terms of threads) are allocated, since DTC creates threads to handle the requests in its queue.

To improve system reliability, the self-adaptive framework in this thesis makes use of the HS/HA concurrency architecture and the LFs concurrency architecture. Initially, HS/HA is used; if it fails, requests are sent to LFs so the requests will still be processed at runtime without having to reboot the system.

To summarize the overall operation of the SAFCA: after a request arrives, the *ArchitectureManager* places the request in the queue specified by the shared object *QueueName*. The arrival time is recorded on the request and the *numOfRequestsReceived* counter is updated by the *ArchitectureManager*. The *ConcurencyArchitecture* that services the queue removes the request from the queue and processes it. The *numOfRequestFinished* counter is updated, the response time of the request is calculated, and the *totalResponseTime* is updated. The *Decider* uses the values of the *numOfRequestsReceived*, the *numOfRequestFinished*, and the *totalResponseTime*, to decide if actions are required. If so, the *Decider* notifies the *Executor* and informs the *Executor* the new queue that will be receiving future request. The *Executor* then modifies the value in *QueueName* accordingly.

Chapter 4 Experiments and Analysis

This chapter describes experiments conducted and presents analysis results, which is organized as follows. First, the chapter describes the experiment settings, experiment parameters and their values used in experiments, and performance measurements taken. Next, the performance of HS/HA under different thread pool sizes is examined in an experiment. A performance comparison of HS/HA and DTC is conducted afterwards. The performance of SAFCA-A, SAFCA-B, and SAFCA-Q is compared with that of DTC and HS/HA. The SAFCA-R reliability analysis is performed next. Finally, all the experiment results are summarized.

Essentially, the experiments demonstrate the idea of self-adaptation to cope with peak demands. Traditionally, systems have been statically configured for worst-case scenarios maximum demands. The main problems with this approach are low resource usage and high cost. The cost could be significantly lowered if the maximum required resources could be reduced, even if the average demands increases. In the performance analysis of SAFCA approaches, the performance of a single concurrency-architecture (DTC or HS/HA) is shown to be not as effective as the adaptive approach, SAFCA, when dealing with bursty workloads. SAFCA incorporates both DTC and HS/HA into the self-adaptive framework: it uses HS/HA for regular normal workloads and adapts to busty workloads

by automatically invoking DTC for bursty workloads. The approach avoids over-provisioning the system and allocates resources as needed. The approach also supports scaling up and down services or applications.

4.1 Experiment Settings

The experiments consider a multi-tier system, as illustrated in Figure 4.1. The server receives requests from multiple clients, and the traffic generated by these multiple clients contains random bursts from time to time. Each request received by the server is processed and a reply is sent back to the client.

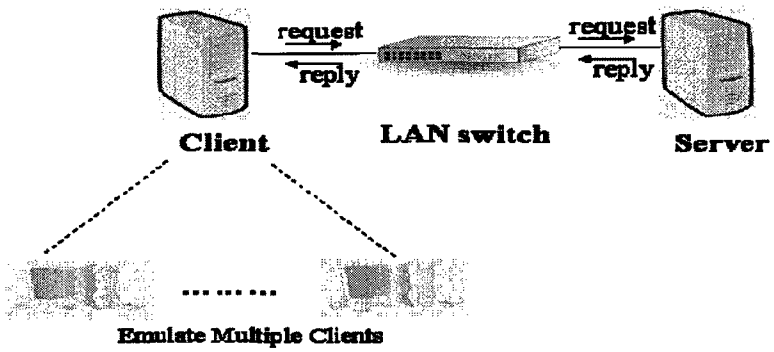

Figure 4.1: Experiment Setup

Our test bed consists of one server machine (3.0 GHz Pentium 4 systems with 3.49 GB of RAM) and a client machine (3.0 GHz Pentium 4 systems with 3.49 GB of RAM) connected to a Phoebe Ethernet Switch (8-Port 10/100Mbps Auto/MDIX). SAFCA is developed with SUN JDK 1.6 as the JAVA platform running on Microsoft Windows XP Professional on the server side. Multiple clients generate traffic and send requests to the server. The client traffic generator is also developed with the same platform as that of the server.

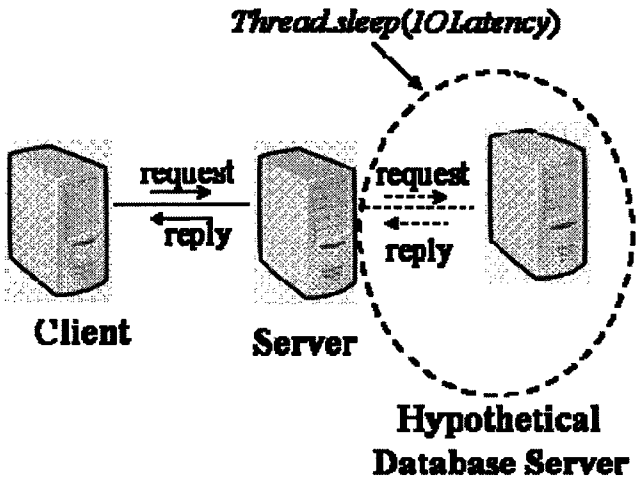

Figure 4.2: I/O Bound Operation in Request Processing

Similar to the experiment in [Oso07], the server machine emulates the application server and the database server in a multi-tier system (also shown in Figure 3.1). As demonstrated in Figure 4.2, the server machine in the test bed acts as the application server that interacts with a hypothetical database server. The database server receives requests and imposes a database access delay before returning a response. This process is emulated on the application server using the Java thread *sleep*() method. Since no CPU cycles are consumed when a thread invokes *sleep*(), this is designated as the I/O bound operation in the processing of a request. Welsh, et al. [Wel00] use a similar emulation approach.

SAFCA can handle random request arrivals of various stochastic distributions. However, this thesis focuses on development of the framework and the performance of different self-adaptive policies. Thus, simple distributions, such as exponential distribution and uniform distributions are used in the experiments. Other random

distributions can also be adopted in the experiments. Multi-clients generate traffic with exponential distribution for inter-arrival time (*interArrivalTime*) between requests. The exponential distribution is realized using the following equation [Ban01] :

$$\text{interArrivalTime} = - \text{meanInterArrivalTime} * \ln(1 - R) \quad (15)$$

R is a uniformly distributed random variable from 0 to 1.

To generate different types of traffic, the *meanInterArrivalTime* value can be one of two values: 1/*normalArrivalRate* or 1/*burstyArrivalRate*. Initially, *interArrivalTime* follows the exponential distribution with a mean of 1/*normalArrivalRate*. After some random period of time, a burst occurs; *interArrivalTime* for bursts is used with a mean of 1/*burstlArrivalRate*. The burst-duration length (*burDuration)* and normal-duration length (*norDuration)* are both uniformly distribute; they are generated using the following equations [Ban01]:

$$\text{burDuration} = (\text{maxBurDuration} - \text{minBurDuration}) * R + \text{minBurDuration} \quad (16)$$

$$\text{norDuration} = (\text{maxNorDuration} - \text{minNorDuration}) * R + \text{minNorDuration} \quad (17)$$

R is a uniformly distributed random variable from 0 to 1.

The client machine and the server machine communicate with each other via Java User Datagram Protocol (UDP) sockets. Transmission Control Protocol (TCP) socket is not used, because TCP is a connection-based protocol with built-in congestion mechanism, while UDP is a connection-less protocol. With UDP, traffic is more easily controlled than TCP for experiments. Each UDP datagram size is 1kB.

4.2 Performance Measurements

This section provides the definitions of all performance measurements that are collected on the server side.

i. **Arrival Rate (*arrivalRate*).** The average number of requests received by the server per second.

ii. **Response Time (*responseTime*).** The average time a request spends in the system, from the time it is received by the server to the time it has been processed completely by the server.

iii. **Request Drop Ratio (*requestDropRatio*).** The number of requests received but dropped divided by the total number of requests received.

iv. **System Throughput (*throughput*).** The number of requests processed per second.

v. **Thread Pool Utilization (*threadPoolUtilization*).** The average number of threads concurrently running divided by the total number of threads created in the thread pool. Some threads may not be used if the workload is not high.

vi. **CPU Utilization (*cpuUtilization*).** The CPU utilization is measured using the built-in *Performance Logs and Alerts* tool from Microsoft Window (XP Professional) [Ms06].

4.3 Experiment Parameters

Three variations of SAFCA (SAFCA-A, SAFCA-B, and SAFCA-Q) are evaluated against HS/HA-only and DTC-only architectural alternatives. SAFCA-R is compared against HS/HA-only architecture. A number of experiments with different settings have

been conducted. A representative set of results are demonstrated in later sections. The following are the parameters used to control the experimental system.

Experiment Parameters for the Server Application:

i. **Thread Pool Size (*threadPoolSize*)**. The total number of threads created in the thread pool. This parameter is used in HS/HA, SAFCA-A, SAFCA-B, SAFCA-Q, and SAFCA-R.

ii. **Concurrency Architecture Queue Size (*queueSize*)**. The maximum capacity of a concurrency queue. This parameter is used in HS/HA, DTC, SAFCA-A, SAFCA-B, SAFCA-Q, and SAFCA-R.

iii. **Sampling Interval Length (*samplingIntervalLength*)**. The *Monitor* collects performance data at the end of each sampling interval. This parameter is used in HS/HA, DTC, SAFCA-A, SAFCA-B, SAFCA-Q, and SAFCA-R.

iv. **I/O Bound Operation Latency Time (*IOLatency*)**. The sleep duration parameter is used when invoking Java thread *sleep()* method to emulate database server interaction. This parameter is used in HS/HA, DTC, SAFCA-A, SAFCA-B, SAFCA-Q, and SAFCA-R.

v. **Switch Back Queue Length Threshold (*queueInvThreshold*)**. When the length of a concurrency architecture queue falls below the Queue Length Inverse Threshold, the *Decider* notifies the *Executor* to divert requests back to this queue. This parameter is used in SAFCA-Q.

vi. **Burst Detection Arrival Rate Percentage (*x*).** If the current arrival rate is greater than *x*% of the previous arrival rate, then a burst is assumed to have occurred. On the other hand, if the current arrival rate is less than the previous one and the difference is greater than *x*%, then the burst is assumed to have finished. This parameter is used in SAFCA-A.

vii. **Burst Detection Response Time Percentage (*y*).** If the difference between the current response time and previous response time also jumps by *y*%, then the *Decider* notifies the *Executor* to take actions. This parameter is used in SAFCA-A.

Experiment Parameters for the Client Application:

i. **Burst Arrival Rate (*burstyArrivalRate*).** The average number of requests sent per second during a burst period.

ii. **Normal Arrival Rate (*normalArrivalRate*).** The average number of requests sent per second during a normal period.

iii. **Maximum Burst Duration (*maxBurDuration*).** The maximum number of sampling intervals in a burst period.

iv. **Minimum Burst Duration (*minBurDuration*).** The minimum number of sampling intervals in a burst period.

v. **Maximum Normal Duration (*maxNorDuration*).** The maximum number of sampling intervals in a normal period.

vi. **Minimum Normal Duration (*minNorDuration*).** The minimum number of sampling intervals in a normal period.

Some parameters in our experiments are fixed, as shown in Table 4.1. The burst arrival rate and the normal arrival rate are selected such that their difference is significant. The min/max parameters for the burst duration and normal duration are selected to achieve a good balance of normal time periods and burst time periods in an experiment run. Specifically, under a request arrival rate of 200 requests per second, the system utilization is around 80% which is considered very high. Under a request arrival rate of 50 requests per second, the system utilization is around 20%. The arrival rate for a burst period is set at 200 requests per second and the arrival rate for a normal period is set at 50 requests per second to avoid reaching system hardware resource bottleneck. If the hardware resource bottleneck is reached, the performance of the system could not be improved using SAFCA. The *burstyArrivalRate* and *normalArrivalRate* can be set to other values depending on the experiment environment. The bursty and normal period durations can also take on any value.

Table 4.1: Experiment Parameters

Experiment Parameters	Value
burstyArrivalRate	200 requests/sec
normalArrivalRate	50 requests/sec
maxBurDuration	20 intervals
minBurDuration	10 intervals
maxNorDuration	30 intervals
minNorDuration	20 intervals
samplingIntervalLength	5 sec
IOLatency	1 sec

4.4 Performance Evaluation of HS/HA

HS/HA extends the thread pool concurrency architecture. The fixed thread pool size, in terms of the number of threads created, needs to be pre-configured. Setting this value too high can lead to low resource utilization, while setting this value too low can lead to a long response time and a high request drop ratio.

The response time, the request loss ratio, and the thread pool utilization for different thread pool sizes of HS/HA have been measured over a number of experiments. In these experiments, the queue size of HS/HA is configured to be 50 requests.

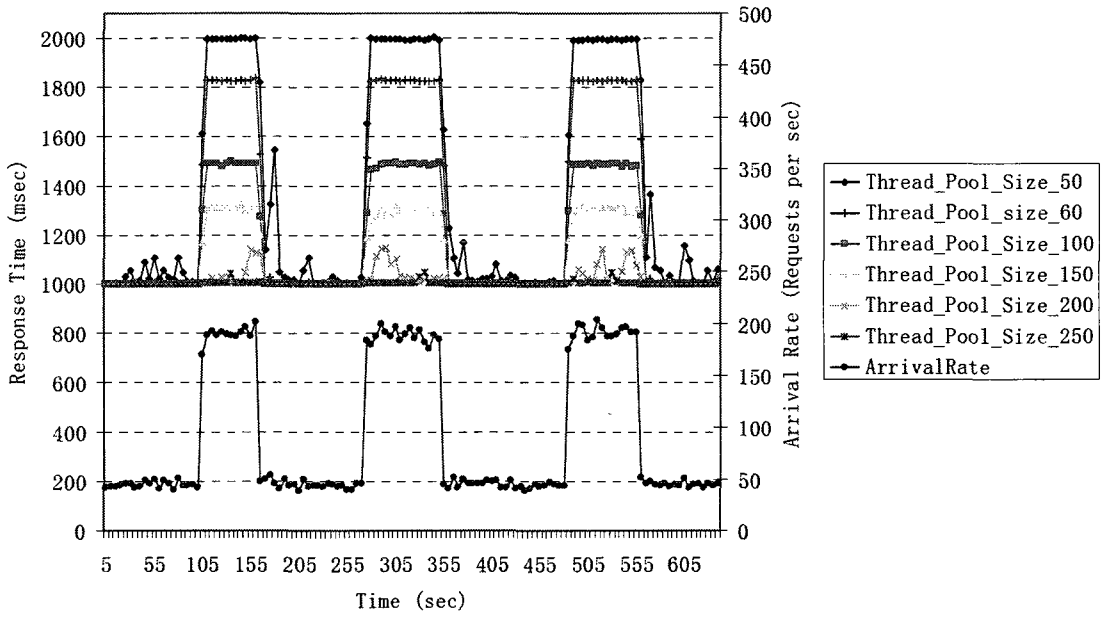

Figure 4.3: Response Time versus Different Thread Pool Sizes for HS/HA

Figure 4.3 plots the response times of different thread pool sizes (50, 60, 100, 150, 200, 250 number of threads) for HS/HA on the primary y-axis (on the left) and the arrival rate on the secondary y-axis (on the right). There are three bursts of workload in this

figure, occurring at random times with random durations. During burst periods, the average number of requests per second is around 200, while it is around 50 requests per second for non-burst periods. The response time for a thread pool of size 50 is the highest. Setting the pool size to 60 reduces the response time by a small amount. At size 250, the response times for HS/HA during burst periods and non-burst periods are the same. Therefore, the number of threads in the thread pool should be greater than 200 in order for the response time not to be affected by the burst. For normal workloads, only 60 threads are needed in the thread pool for the response time to be around 1 sec.

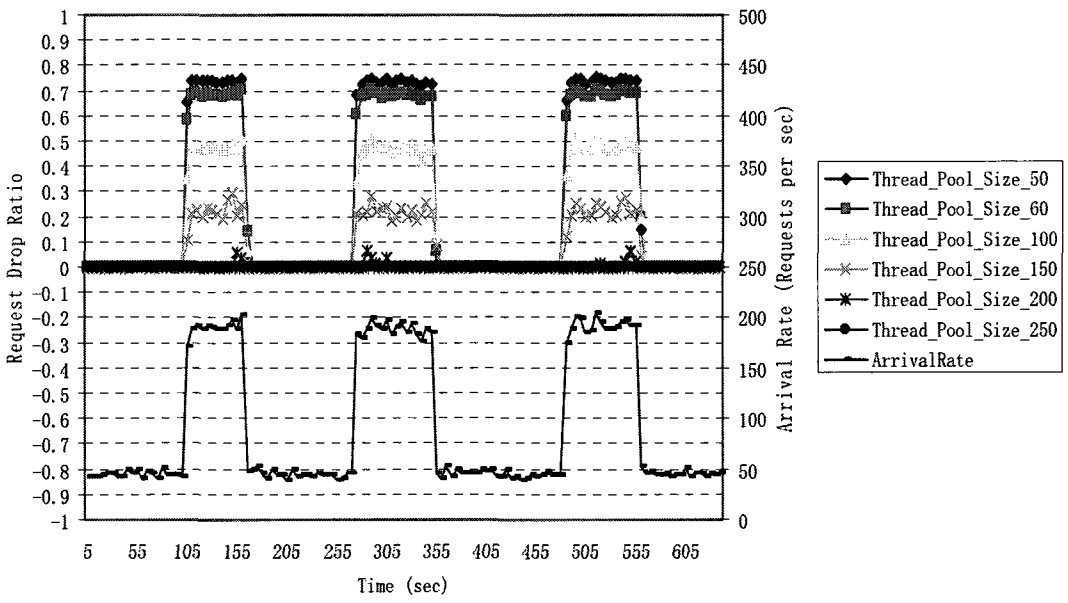

Figure 4.4: Request Drop Ratio versus Different Thread Pool Sizes for HS/HA

Figure 4.4 plots the request drop ratios versus different thread pool sizes (50, 60, 100, 150, 200, 250 number of threads) for HS/HA on the primary y-axis (on the left) and the arrival rate on the secondary y-axis (on the right). The arrival rates used in the

experiments are the same as those presented in Figure 4.3. The drop ratio for a thread pool of size 50 is the highest at about 75%. Setting the pool size to 60 reduces the drop ratio by a small amount to around 70%. At size 250, the drop ratio for HS/HA during burst periods and non-burst periods are both around 0. Therefore, the number of threads in the thread pool should be greater than 200 in order for the drop ratio not to be affected by the burst. For normal workloads, only 60 threads are needed in the thread pool for the drop ratio to be around 0.

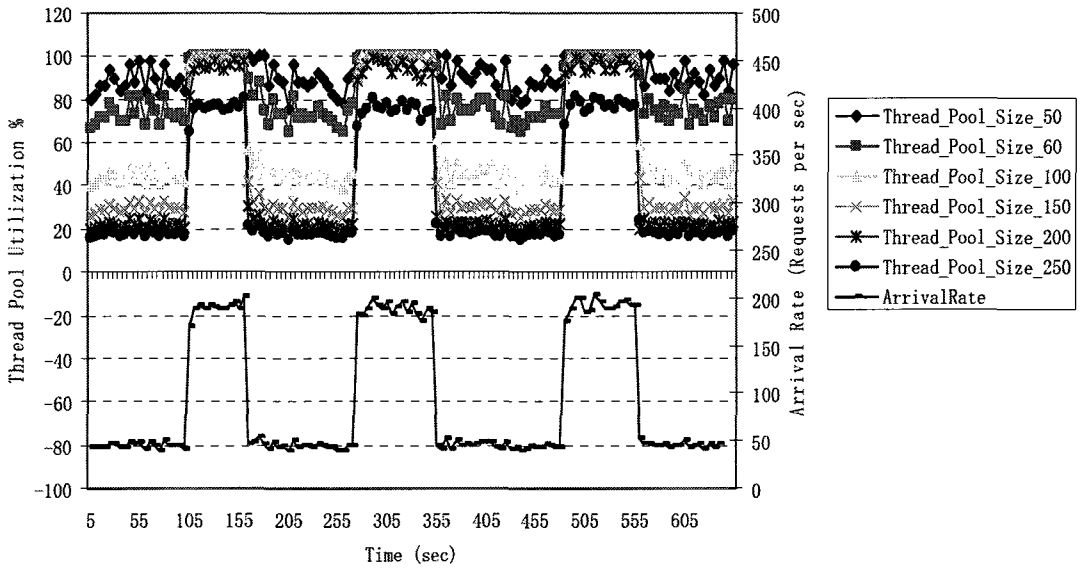

Figure 4.5: Thread Pool Utilization versus Different Thread Pool Sizes for HS/HA

Figure 4.5 shows the thread pool utilization of different thread pool sizes (50, 60, 100, 150, 200, 250 number of threads) for HS/HA on the primary y-axis (on the left) and the arrival rate on the secondary y-axis (on the right). There are three bursts of workload in this figure, occurring at random times with random durations. During non-burst periods,

the thread pool utilization for a thread pool of size 50 is highest ranging from 80% to 100%. Setting the pool size to 60 reduces the utilization by a small amount from about 70% to 80%. At size 250, the utilization is only around 25%. When a burst occurs, the utilization will be around 80% for thread pool sizes from 50 to 250. For normal workloads, only 60 threads are needed in the thread pool for the thread pool utilization to be around 80% which is often considered as a situation point.

Results in this section demonstrate that, during normal workloads, the performance is better for smaller pool sizes. This is because while a large thread pool can handle a high-burst of requests, many of the threads are idle during normal conditions, since there are only a few requests. Many systems today are configured to handle worst-case scenarios such as high-burst workloads. However, a lot of resources are wasted during non-burst periods. Based on the results, the thread pool size is to be determined at 60 threads for SAFCA, because the results show that this value can handle the normal request rate and produce acceptable performance.

4.5 Performance Evaluation of HS/HA and DTC

This section compares the performances of the HS/HA and DTC concurrency architectures under bursty workloads. The parameters used for HS/HA in this experiment are: *threadPoolSize* is 60 and *queueSize* is 50. The *queueSize* of DTC is set to 50 as well.

Figure 4.6 demonstrates that during burst periods, the response time of HS/HA (around 1800ms) is much larger than that of DTC (around 1000 ms). Also, the thread utilization of DTC is always 100%, since it creates a new thread for each new request. After the request is finished, the thread dies. However, if one focuses on the non-burst

periods and changes the primary y-axis scale, the figure show that the response time for HS/HA is shorter than that of DTC. This is shown in Figure 4.7.

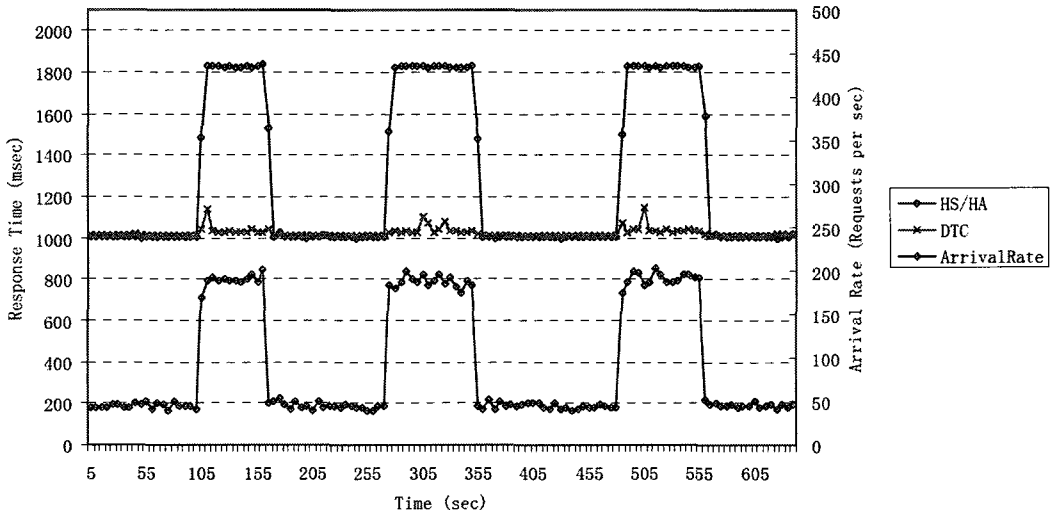

Figure 4.6: Comparison of Response Time for HS/HA and DTC

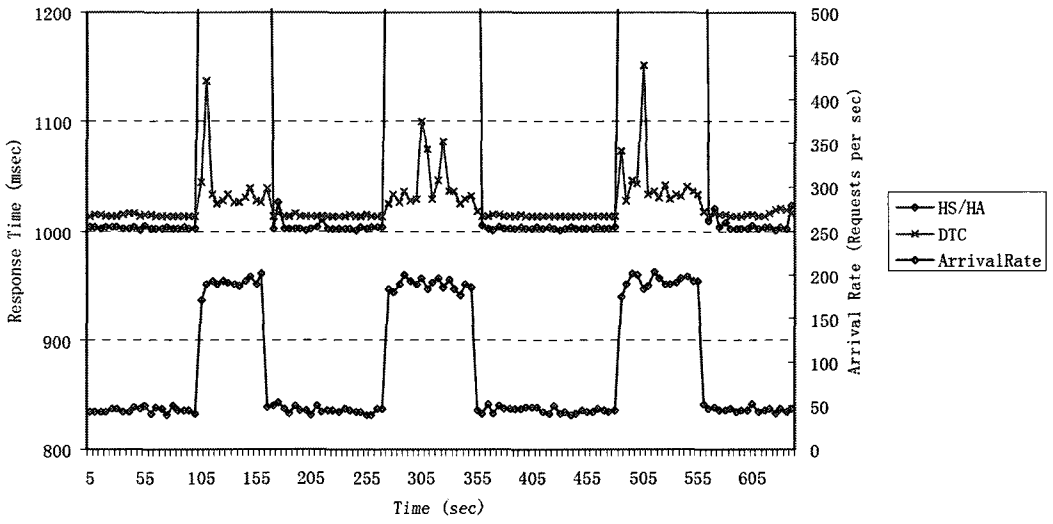

Figure 4.7: Comparison of Response Time for HS/HA and DTC with Finer Y-Axis Scale

During normal workload conditions, the response time of HS/HA is smaller because it does not incur thread creation/destruction overhead. During burst periods, DTC can create more threads to reduce response time. Figure 4.7 also shows jumps in the response time for DTC. This may be caused by the garbage collector in the Java virtual machine. For these reasons, SAFCA is proposed, which enables self-adaptation in response to dynamic traffic loads. As stated in Section 3.3, SAFCA contains both HS/HA and DTC. Normally, HS/HA processes requests from the HS/HA queue. If a burst arrives, requests will be put to the DTC queue instead, so more resources (in terms of threads) are allocated, since DTC creates threads to handle the requests in its queue. Several variations of SAFCA have been proposed and evaluated in the following sub-sections.

4.6 Performance Evaluation of SAFCA-Q

As described in Section 3.6.1, for SAFCA-Q, when the self-adaptive mechanism has not been triggered, new requests are placed in the HS/HA queue. When the HS/HA queue becomes full, new requests are placed in the DTC queue. Once the HS/HA queue length has dropped below *queueInvThreshold*, new requests are placed in the HS/HA queue. This section presents comparison of the performance of SAFCA-Q with HS/HA and DTC, in terms of response time, request drop ratio, and CPU utilization. The performance of SAFCA-Q with different queue sizes and different *queueInvThreshold* values are also evaluated. SAFCA contains both the HS/HA concurrency architecture and the DTC concurrency architecture, thus the parameters of SAFCA includes the queues sizes for both HS/HA and DTC. The parameters are recorded in Table 4.2. The *queueSize* values

are fixed in this experiment. The experiment in Section 4.6.3 involves varying the *queueSize* value.

Table 4.2: SAFCA-Q Parameters

Experiment Parameters	Value
queueSize(HS/HA)	25 requests
queueSize(DTC)	25 requests
threadPoolSize	60
queueInvThreshold	10

4.6.1 Performance Evaluation of SAFCA-Q and HS/HA

In this experiment, the *queueSize* of HS/HA is set to 50 and the *threadPoolSize* is set to 60. The response time, the request drop ratio, and the CPU utilization are measured for evaluation. The results show that SAFAC-Q offers better performance.

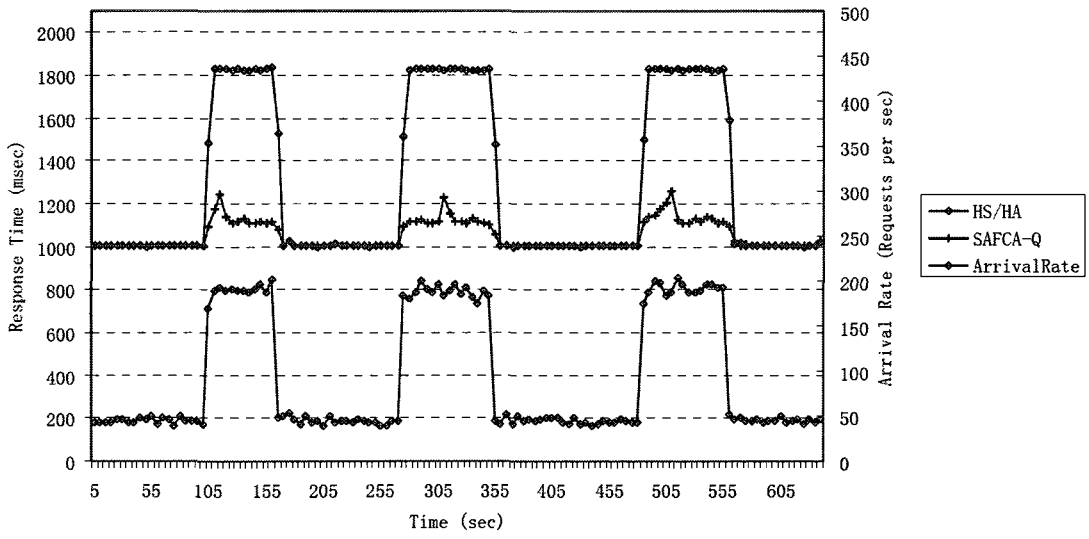

Figure 4.8: Comparison of Response Time for SAFCA-Q and HS/HA

Figure 4.8 presents the response times of SAFCA-Q and HS/HA on the primary y-axis (on the left) and the arrival rate on the secondary y-axis (on the right). The response time of SAFCA-Q is low (around 1000 msec) during normal workloads, beacause it uses HS/HA which has good performance as described Section 4.4. Even during bursts, the response time for SAFCA-Q is still low, because SAFCA-Q invokes DTC to cope with high demands. After a burst is over, the resources for DTC will be released.

Figure 4.9 illustrates the request drop ratios of for SAFCA-Q and HS/HA on the primary y-axis (on the left) and the arrival rates on the secondary y-axis (on the right). The drop ratio of SAFCA-Q is close to 0 during both normal workloads and burst workloads, because SAFCA-Q dynamically allocates more resources during burst periods.

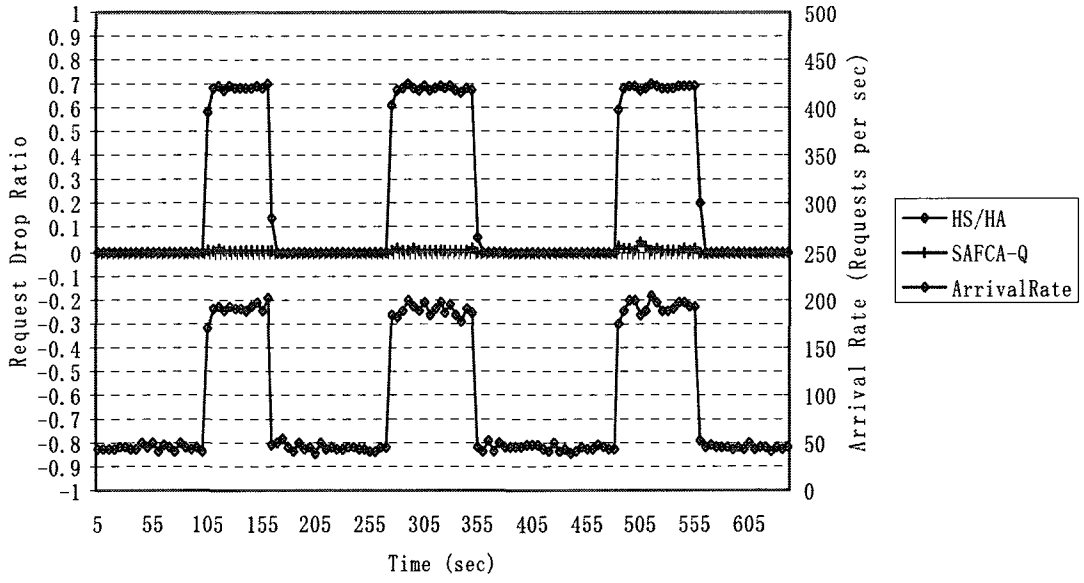

Figure 4.9: Comparison of Drop Ratios for SAFCA-Q and. HS/HA

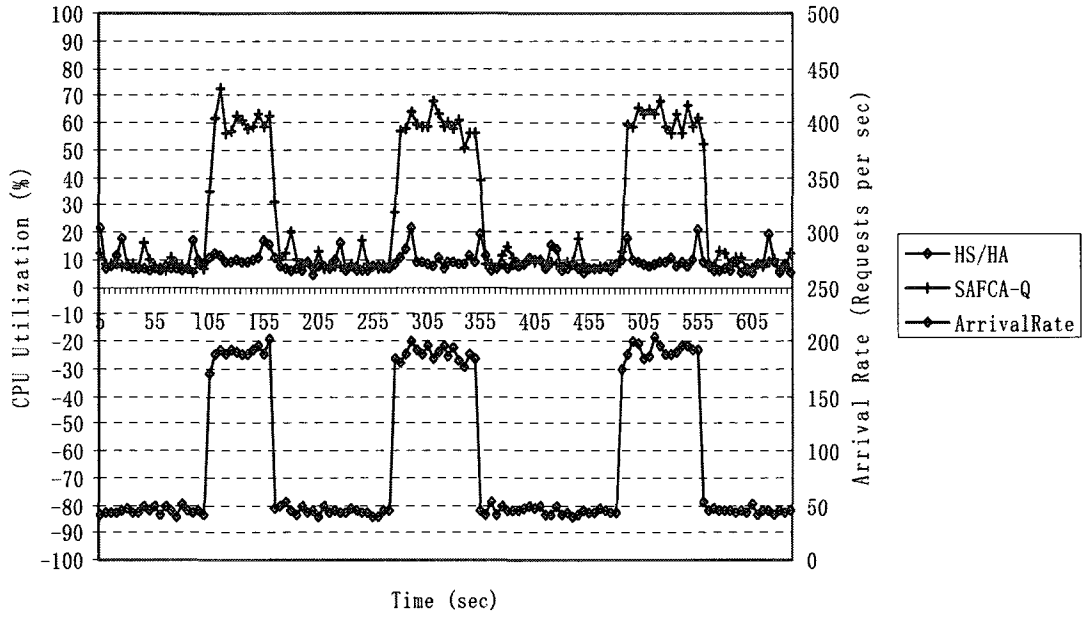

Figure 4.10: Comparison of CPU Utilization for SAFCA-Q and HS/HA

Figure 4.10 demonstrates the CPU utilization for SAFCA-Q and HS/HA on the primary y-axis (on the left) and the arrival rate on the secondary y-axis (on the right). The CPU utilization is always low (between 10% and 20%) for stand-alone HS/HA, because the thread pool size is fixed. However, because SAFCA-Q uses DTC to create more threads during bursts, the CPU is better utilized with utilization in the range of 60% to 70%.

4.6.2 Performance Evaluation of SAFCA-Q and DTC

In this experiment, the response time, the request drop ratio, the CPU utilization, and the number of created threads are measured for this performance evaluation. The results

show that SAFAC-Q offers better performance results in most cases and similar performance results in some cases.

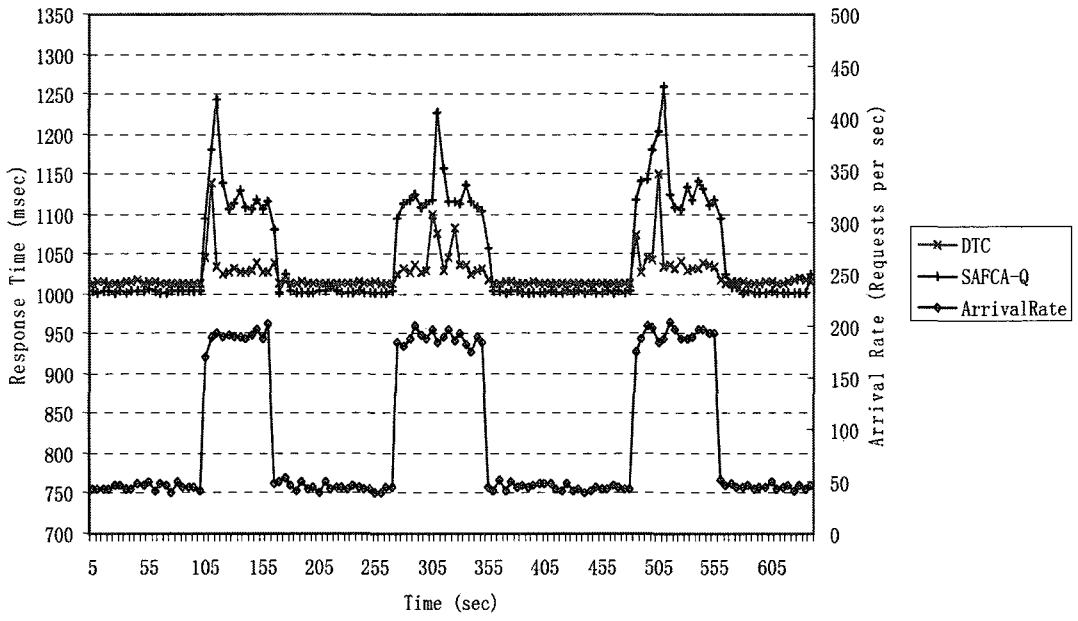

Figure 4.11: Comparison of Response Time for SAFCA-Q and DTC

Figure 4.11 depicts the response times of for SAFCA-Q and DTC on the primary y-axis (on the left) and the arrival rate on the secondary y-axis (on the right). In most environments, non-bursty periods are much longer than bursty periods and the figure shows that the response time of SAFCA-Q (around 1000 msec) is better than that of DTC (around 1020 msec) during normal workload. During bursts, the response time of SAFCA-Q is a little longer than that of DTC. However, as Section 4.6.3 demonstrates, changing the *queueSize* in SAFCA-Q will decrease the response time.

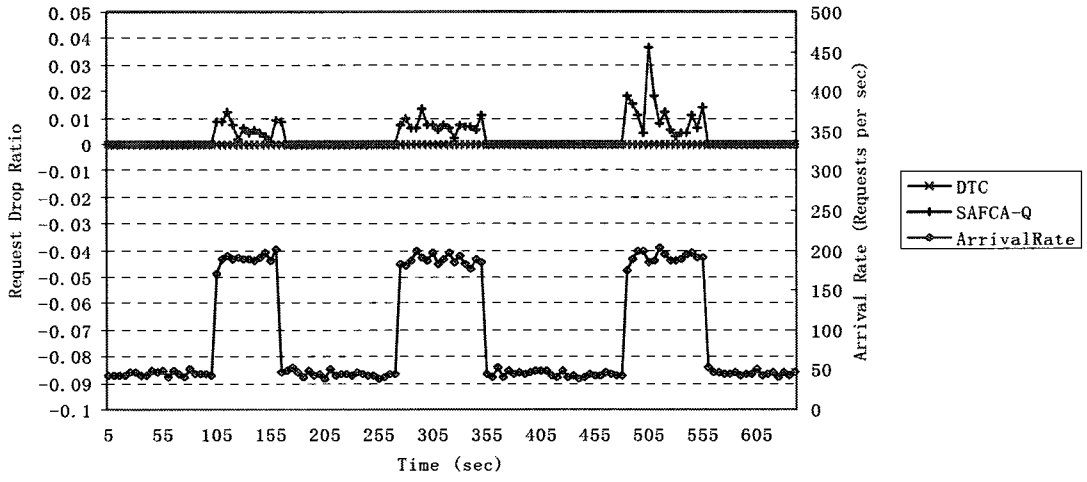

Figure 4.12: Comparison of Drop Ratios for SAFCA-Q and DTC

Figure 4.12 demonstrates the loss ratio of for SAFCA-Q and DTC on the primary y-axis (on the left) and the arrival rate on the secondary y-axis (on the right). Both SAFCA-Q and DTC has a loss ratio close to 0 during non-burst periods. When in a burst, SAFCA-Q has a slightly larger loss ratio than DTC, but the ratio is small (less than 0.05).

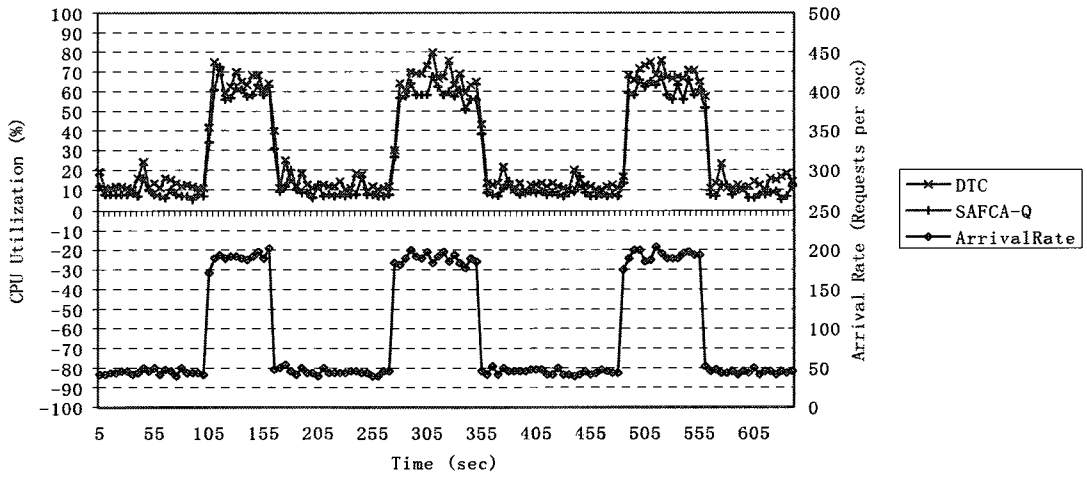

Figure 4.13: Comparison of CPU Utilization for SAFCA-Q and DTC

Figure 4.13 shows that SAFCA-Q has similar CPU utilization as DTC in both normal workload and burst workload conditions.

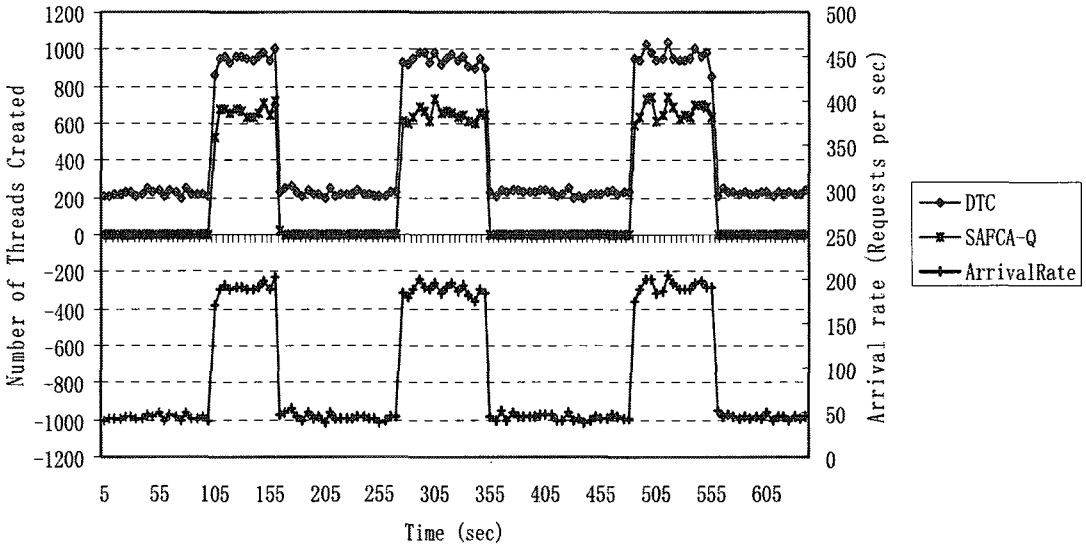

Figure 4.14: Comparison of Number of Threads Created for SAFCA-Q and DTC

In terms of resource usage, Figure 4.14 illustrates that DTC creates more threads and takes up more resources than SAFCA-Q during normal workload condition. Each thread created consumes memory space and CPU cycles. Further, the operating system will incur the thread creation/destruction overhead. As depicted in the Figure 4.14, in one sampling interval, DTC creates about 200 new threads under normal conditions. In comparison, instead of creating new threads, only 60 existing threads in the thread pool of SAFCA-Q are enough to handle the normal workload. Since normal workload periods are typically much longer than burst periods, SAFCA-Q is the better than DTC from the resource management perspective.

4.6.3 Performance Evaluation of SAFCA-Q with Different Queue Sizes

As stated in Section 3.6.1, in the SAFCA-Q implementation, if the HS/HA concurrency architecture queue becomes full, requests are diverted to the DTC queue. The effect of various queue sizes on the performance has been investigated.

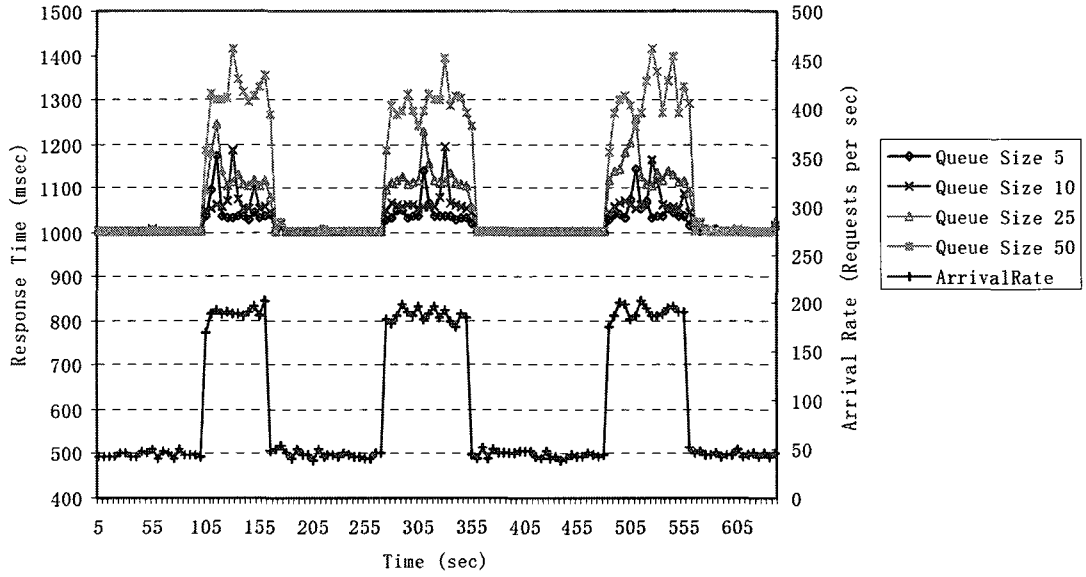

Figure 4.15: Response Time versus Different Queue Sizes for SAFCA-Q

Figure 4.15 demonstrates that, for each different experiment run (each run has a different queue size), during a burst, the queue size has some effect on the response time. For large queue sizes, e.g., 50, the response time is large (around 1300-1400 ms). Smaller queue sizes, e.g., 5 or 10, have smaller response times (around 1000-1100 ms). This is because, the larger the queue, the more requests are stored in the queue during a burst, and hence the waiting time in the queue increases.

4.6.4 Performance Evaluation of SAFCA-Q with Different Threshold

As stated in Section 3.6.1, in the SAFCA-Q design, after a burst, if the HS/HA concurrency architecture queue decreases below a certain threshold (*queueInvThreshold*), requests will diverted back to the HS/HA queue. The effect of this threshold value on performance has been studied.

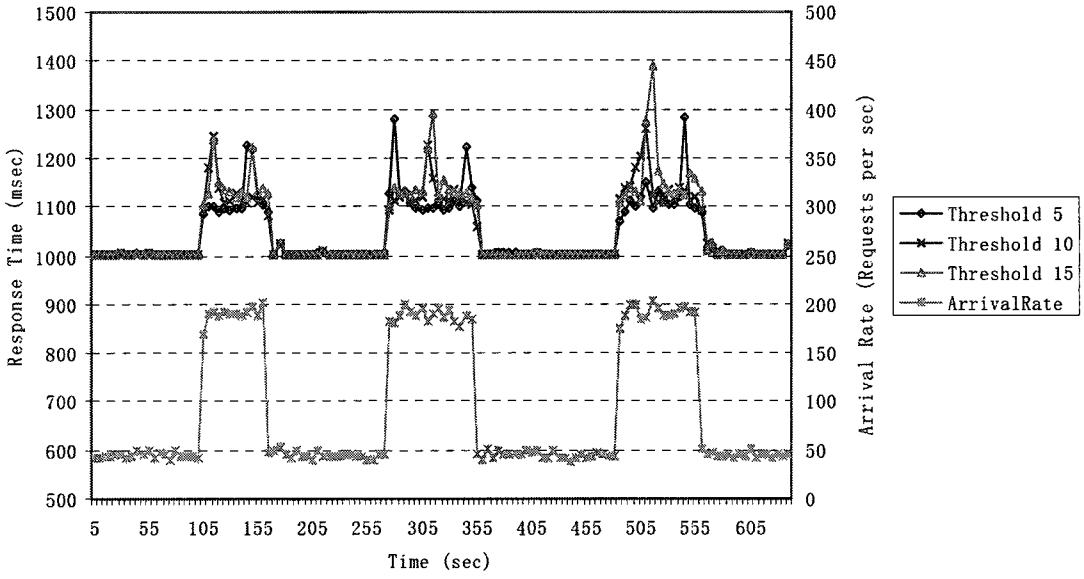

Figure 4.16: Response Time versus Different Threshold for SAFCA-Q

Figure 4.16 illustrates that, during a burst, different threshold values have some effect on the response time. Under normal workload, the response time is about 1000 ms. During a burst, the response time is between 1100 ms to 1300 ms in most cases. In summary, different values have been tested for the threshold so that requests will be diverted back to the HS/HA queue. For example, the HS/HA queue size in SAFCA-Q is set to 25 and the *queueInvThreshold* is set to 10. When the queue becomes full, the

architecture is adapted to DTC while HS/HA continue to work on its queue. Once the HS/HA queue length is reduced below 10, the architecture is adapted back to HS/HA. After trying different values for *queueInvThreshold* and observing no major difference in response time, the thesis simply chose 10 as the value for *queueInvThreshold.*

4.7 Performance Evaluation of SAFCA-A

This section evaluates the performance of SAFCA-A with HS/HA and DTC in terms of response time, request drop ratio, and CPU utilization. As stated in Section 3.6.2, SAFCA-A contains the HS/HA concurrency architecture and the DTC concurrency architecture and initially sends requests to the HS/HA queue. When the arrival rate has increased by more than x% (the value of x is defined in Table 4.3) and the response time also has increased by more than 20%, SAFCA-A sends new requests to the DTC queue. If the arrival rate has decreased by more than x'%, SAFCA-A sends new requests to the HS/HA queue. Table 4.3 summarizes the parameters used for SAFCA-A. The *queueSize* values are fixed in this experiment. The experiment in Section 4.6.3 involves varying the *queueSize* value.

Table 4.3: SAFCA-A Parameters

Experiment Parameters	Value
queueSize(HS/HA)	25 requests
queueSize(DTC)	25 requests
threadPoolSize	60
x	20%

The threshold x has been set to 20% for the subsequent experiment. Section 4.7.3 studies the performance impact for the different Burst Detection Arrival Rate Percentage (x) values.

4.7.1 Performance Evaluation of SAFCA-A and HS/HA

In this experiment, the *queueSize* of HS/HA is set to 50 and the *threadPoolSize* is set to 60. The response time, the request drop ratio, and the CPU utilization are measured for performance evaluation. The results show that SAFAC-A offers better performance.

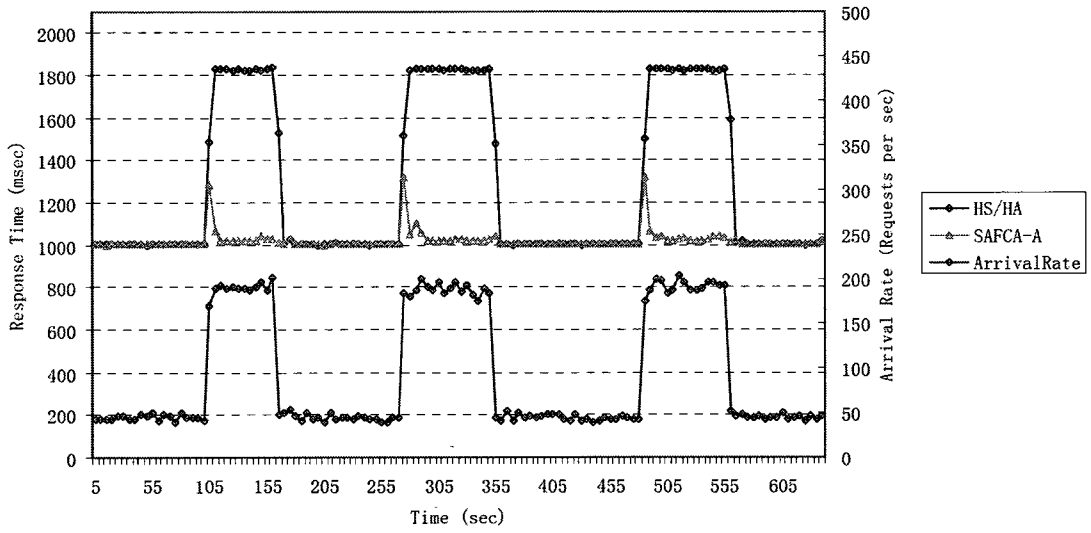

Figure 4.17: Comparison of Response Time for SAFCA-A and HS/HA

Figure 4.17 illustrates the response times for SAFCA-A and HS/HA on the primary y-axis (on the left) and the arrival rate on the secondary y-axis (on the right). The response time of SAFCA-A is low during normal workload because it uses HS/HA. During bursts, the response time is still low because SAFCA-A dynamically makes use of DTC for higher demands.

Figure 4.18 demonstrates the request drop ratio of for SAFCA-A and HS/HA on the primary y-axis (on the left) and the arrival rate on the secondary y-axis (on the right). The drop ratio of SAFCA-A is close to 0 (except the short period when the burst just starts) during both normal workload and burst workload, because it uses HS/HA during non-burst periods and DTC during burst periods. The reason being that the existing number of threads in HS/HA is sufficient to handle the workload during non-burst periods. Since no new threads need to be created, the lack of creation overhead makes HS/HA better than DTC. Similarly, the reason to use DTC to accommodate high demands during burst periods so more threads can be created to handle the demand increase. In contrast, for HS/HA, only a fixed number of threads are available to handle the demand.

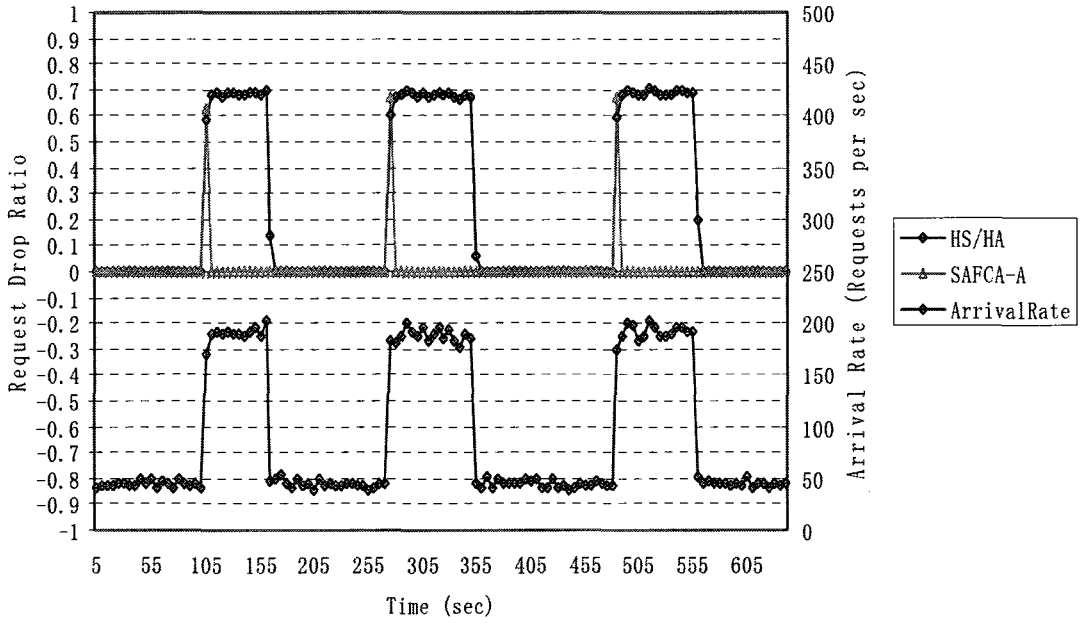

Figure 4.18: Comparison of Drop Ratio for SAFCA-A and HS/HA

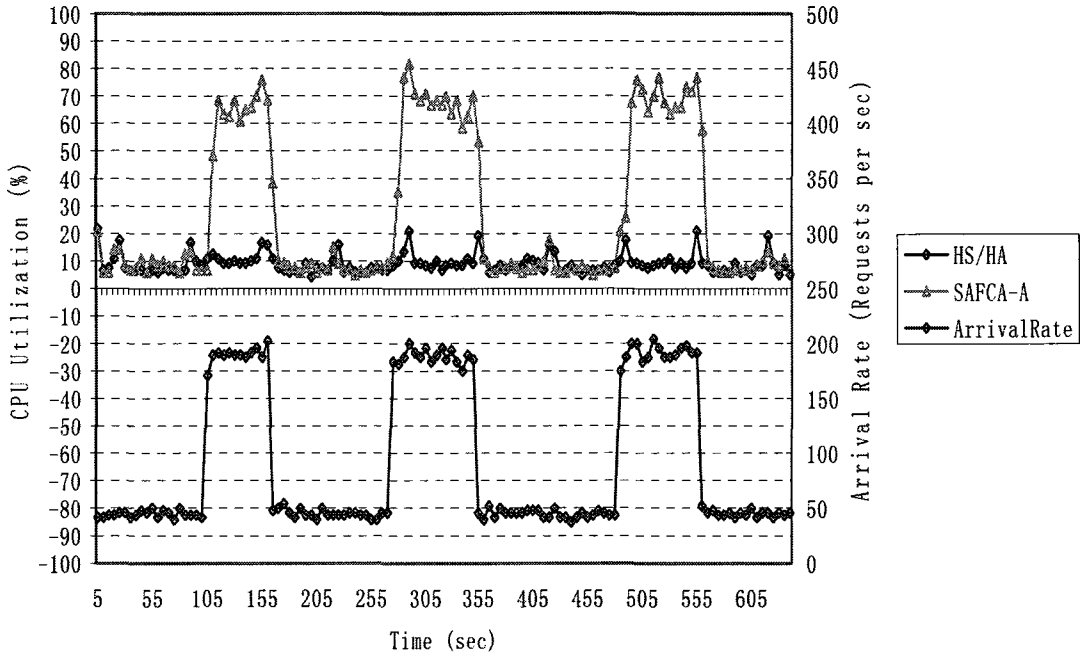

Figure 4.19: Comparison of CPU Utilization for SAFCA-A and HS/HA

Figure 4.19 depicts the CPU utilization for SAFCA-A and HS/HA on the primary y-axis (on the left) and the arrival rate on the secondary y-axis (on the right). The CPU utilization is always low (between 10% and 20%) for stand-alone HS/HA, because the thread pool size is fixed and it does not require a lot of CPU resource. However, because SAFCA-A uses DTC to create more thread during bursts, the CPU is better utilized with utilization in the range of 60% to 70%.

4.7.2 Performance Evaluation of SAFCA-A and DTC

In this experiment, the response time, the request drop ratio, the CPU utilization, and the number of created threads are measured for performance evaluation. The results show

that SAFAC-A results in better performance in most cases and similar performance in some cases.

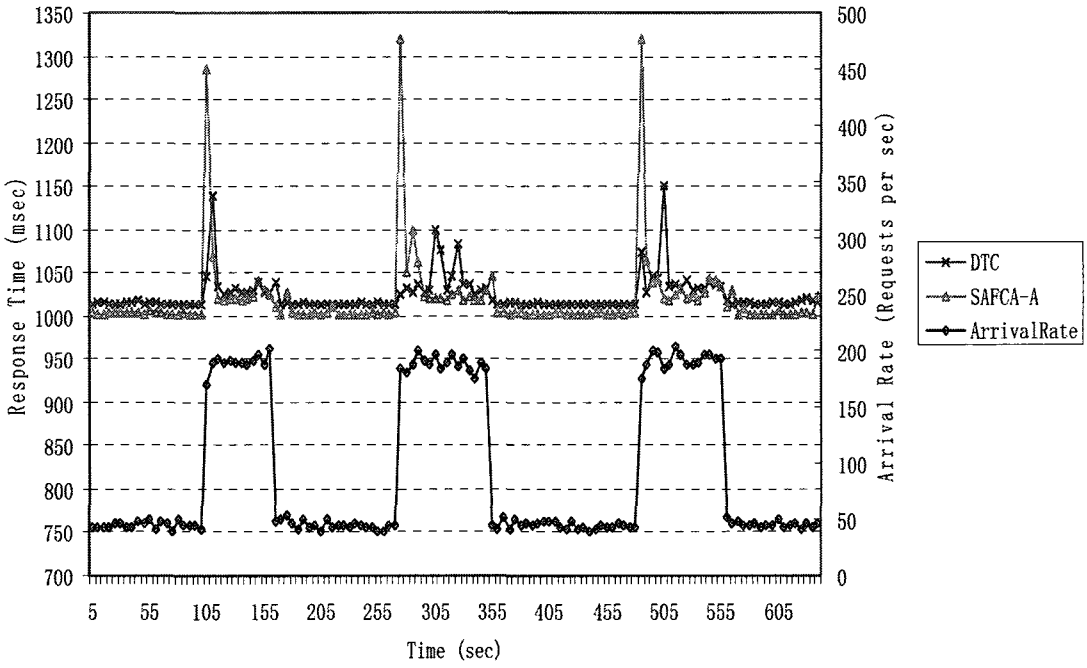

Figure 4.20: Comparison of Response Time for SAFCA-A and DTC

Figure 4.20 demonstrates the response times of for SAFCA-A and DTC on the primary y-axis (on the left) and the arrival rate on the secondary y-axis (on the right). Except when the burst initially started, SAFCA-A has a lower response time than that of DTC. The sampling interval is 5 seconds, and the self-adaptive action can only occur at the end of a sampling interval. Therefore, the response time of SAFCA-A can reach around 1300 msec before the switch. Once the switch takes place, the response time immediately decreases to around 1020 msec.

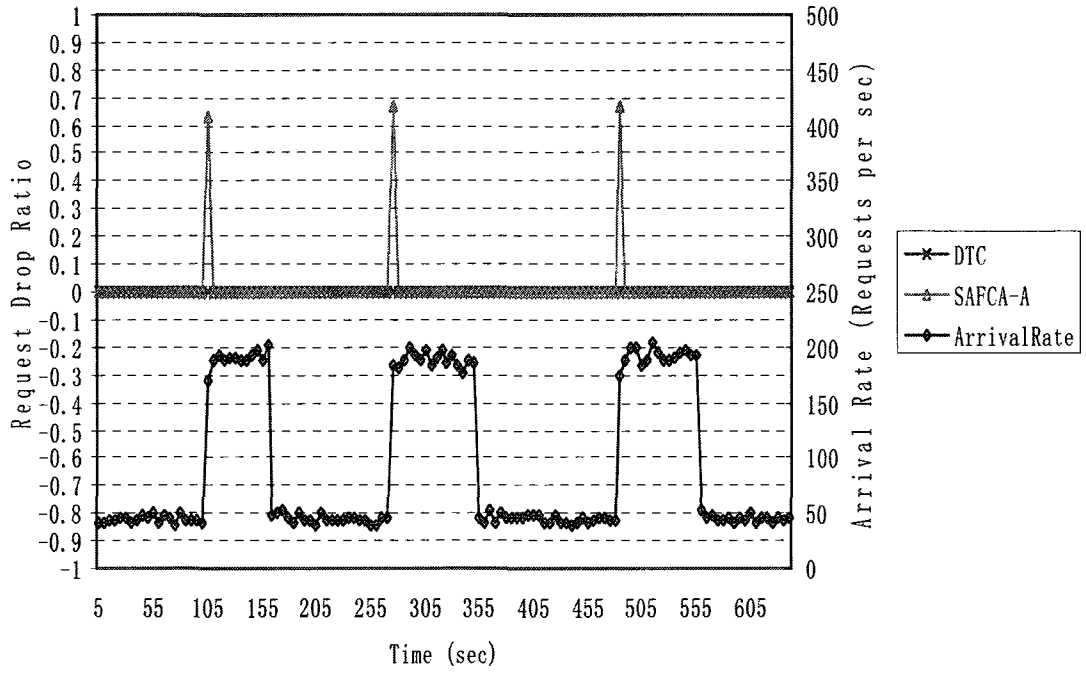

Figure 4.21: Comparison of Drop Ratio for SAFCA-A and DTC

Figure 4.21 depicts the loss ratio for SAFCA-A and DTC on the primary y-axis (on the left) and the arrival rate on the secondary y-axis (on the right). Both SAFCA-A (except when the burst first starts) and DTC have a loss ratio close to 0. The spike in Figure 4.21 is due to the facts that the self-adaptive action can only occur at the end of a sampling interval, therefore, the drop ratio of SAFCA-A can reach around 65% before the switch. Once the switch takes place, the drop ratio immediately decreases to around 0%.

Figure 4.22 demonstrates that SAFCA-A has similar CPU utilization as DTC in both normal workload and burst workload conditions.

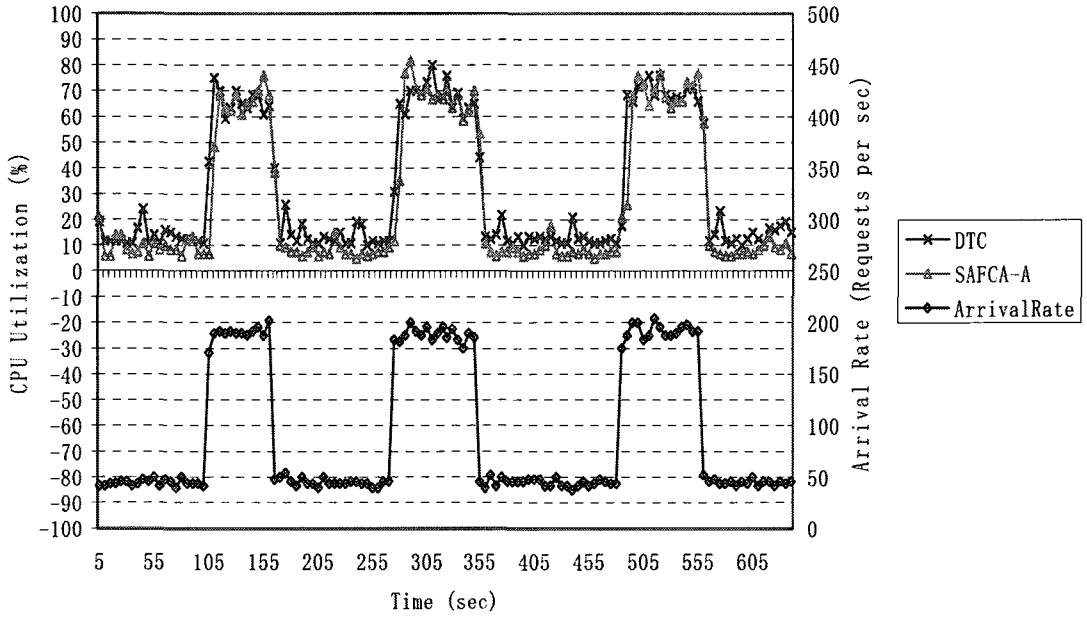

Figure 4.22: Comparison of CPU Utilization for SAFCA-A and DTC

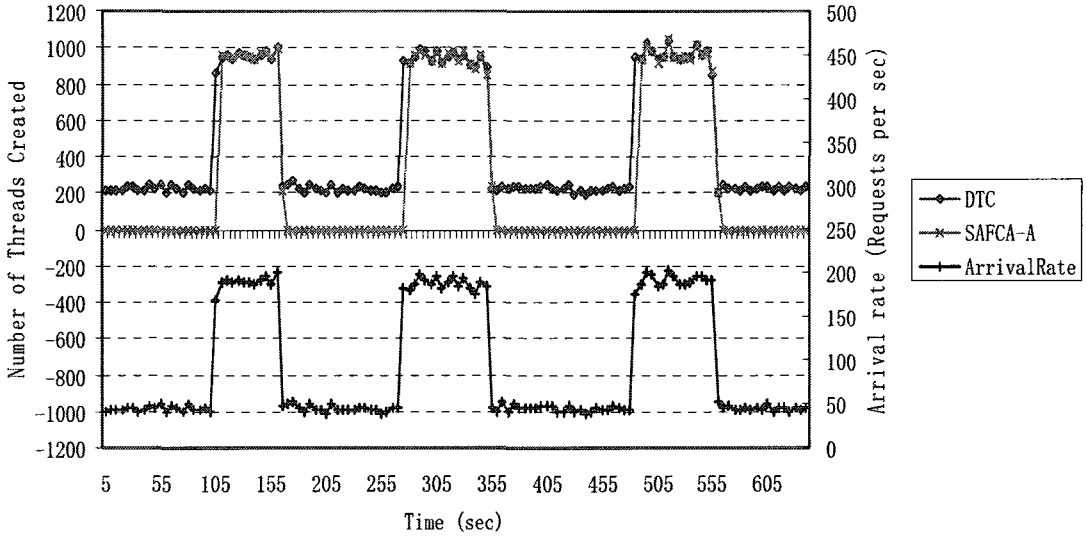

Figure 4.23: Comparison of Number of Created Threads for SAFCA-A and DTC

Figure 4.23 demonstrates that DTC creates more threads and takes up more resources than SAFCA-A during normal workload conditions. Each thread created occupies memory space, CPU cycles, and the operating system incurs the thread creation/destruction overhead. As shown in Figure 4.23, in one sampling interval, DTC creates about 200 new threads under normal conditions. In comparison, instead of creating new threads, only 60 threads in the thread pool of SAFCA-A are used to handle the normal workload. Since normal workload periods are typically much longer than burst periods, SAFCA-A is more efficient in resource usage than DTC.

4.7.3 Performance Evaluation of SAFCA-A with Different Thresholds

As stated in Section 3.6.2, in the SAFCA-A design, if the current arrival rate is less than the previous arrival rate, and their difference is greater than *x*'% of the previous arrival rate, then the burst is assumed to be over. The effect of this parameter has on the performance has been studied.

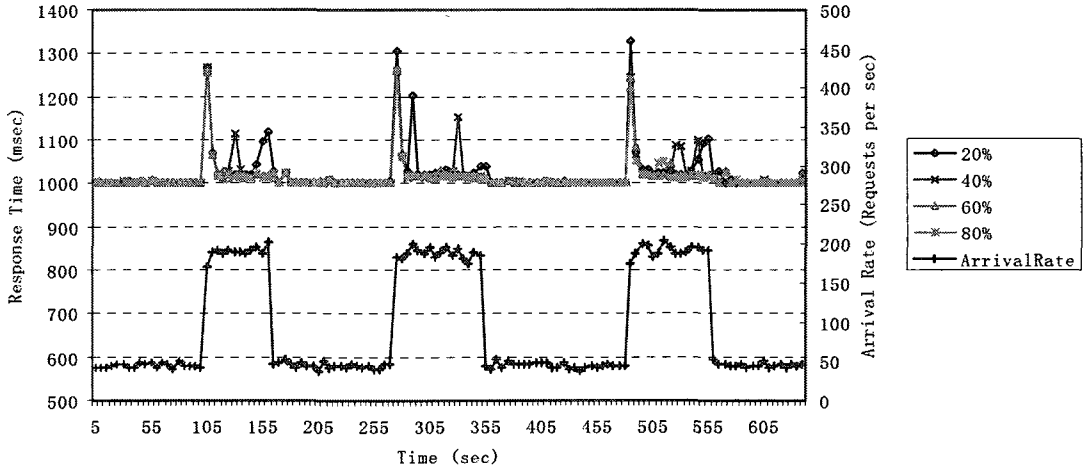

Figure 4.24: Response Time versus Different Thresholds for SAFCA-A

Figure 4.24 illustrates that, during burst, the x value has a negligible effect on the response time. In our experiment, there are only two average arrival rates, normal and burst; as their difference is significant, the value of x has little influence on system performance. For this reason, 20% is chosen as the value for x. Setting the value smaller than 20% may cause oscillation due to the statistical variation of the demand during normal-periods. As stated in Section 3.6.2, burst detection is not the focus of this work.

4.8 Performance Evaluation of SAFCA-B

This section evaluates the performance of SAFCA-B with HS/HA and DTC in terms of response time, request drop ratio, and CPU utilization. As described in Section 3.6.2, SAFCA-B initially sends requests to the HS/HA queue. When the arrival rate has increased by more than one standard deviation from the previous average arrival rate and the response time also has increased by more than 20%, SAFCA-B sends new requests to the DTC queue. If the arrival rate has decreased by more than one standard deviation, SAFCA-B sends new requests to the HS/HA queue. Performance of SAFCA-B with different Burst Detection Arrival Rate Percentage (x) values is also investigated and presented in this section.

The parameters experimented for SAFCA-B includes the queue sizes for both HS/HA and DTC. The parameters are recorded in Table 4.4. The *queueSize* values are fixed in this experiment. The experiment in Section 4.6.3 involves varying the *queueSize* value.

Table 4.4: SAFCA-B Parameters

Experiment Parameters	Value
queueSize(HS/HA)	25 requests
queueSize(DTC)	25 requests
threadPoolSize	60

4.8.1 Performance Evaluation of SAFCA-B and HS/HA

In this experiment, the *queueSize* of HS/HA is set at 25 and the *threadPoolSize* is set at 60. The response time, the request drop ratio, and the CPU utilization are measured for the performance evaluation. The results show that SAFAC-B demonstrates better performance.

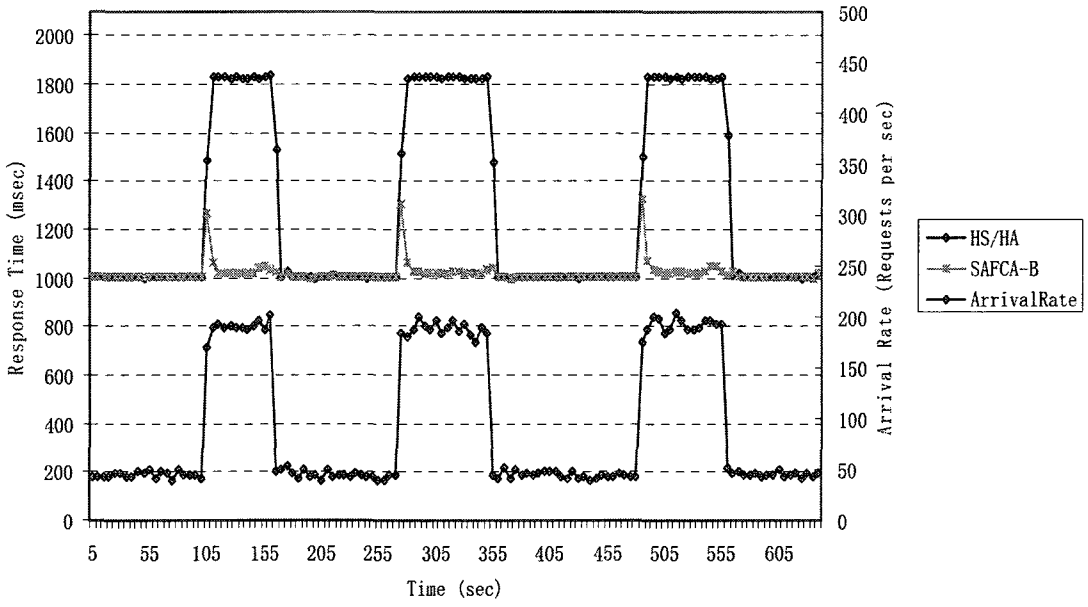

Figure 4.25: Comparison of Response Time for SAFCA-B and HS/HA

Figure 4.25 depicts the response times for SAFCA-B and HS/HA on the primary y-axis (on the left) and the arrival rate on the secondary y-axis (on the right). The response time of SAFCA-B is low during normal workload because it uses HS/HA. During bursts, the response time is still low because SAFCA-B uses DTC instead. This arrangement is used because with HS/HA, the existing number of threads is sufficient to handle the workload during non-burst periods and no new threads are created (no creation overhead). On the other hand, DTC can accommodate high demands during burst periods because it can create more threads. In contrast, for HS/HA, the throughput bottleneck is its fixed number of threads.

Figure 4.26 illustrates the request drop ratio of for SAFCA-B and HS/HA on the primary y-axis (on the left) and the arrival rate on the secondary y-axis (on the right). The drop ratio of SAFCA-B is close to 0 (except when a burst first starts) during both normal workload and burst workload.

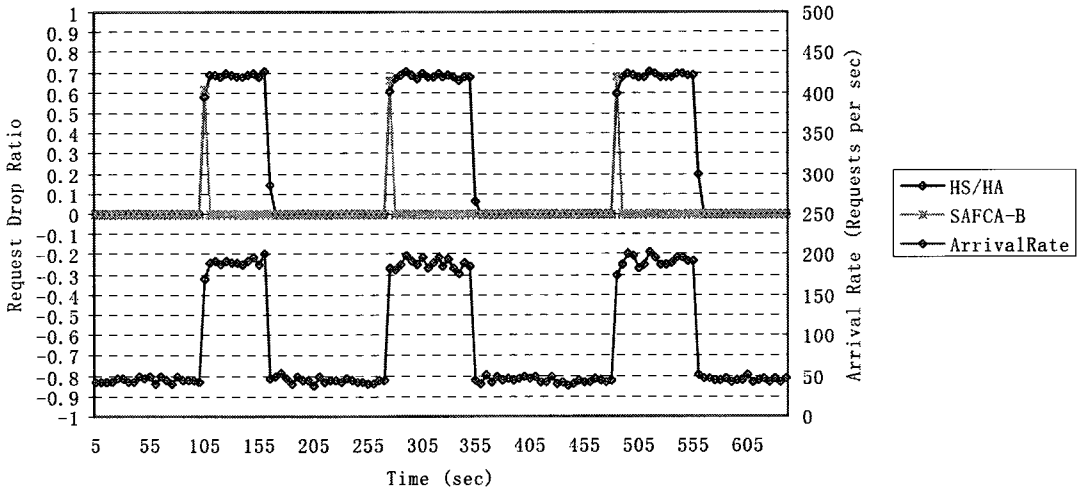

Figure 4.26: Comparison of Drop Ratio for SAFCA-B and HS/HA

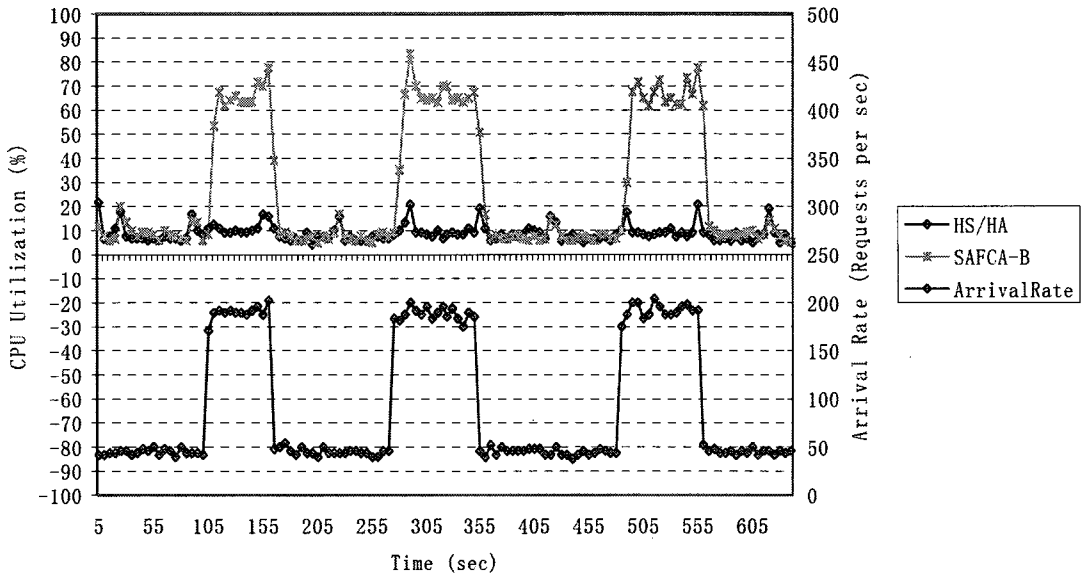

Figure 4.27: SAFCA-B versus HS/HA CPU Utilization Comparison

Figure 4.27 demonstrates the CPU utilization for SAFCA-B and HS/HA on the primary y-axis (on the left) and the arrival rate on the secondary y-axis (on the right). The CPU utilization is always low (between 10% and 20%) for HS/HA because the thread pool size is fixed. However, because SAFCA-B uses DTC to create more threads during bursts, the CPU is better utilized with utilization in the range of 60% to 70%.

4.8.2 Performance Evaluation of SAFCA-B and DTC

In this experiment, the response time, the request drop ratio, the CPU utilization, and the number of created threads are measured for this performance evaluation. The results show that SAFAC-B has better performance in most cases and similar performance in some cases.

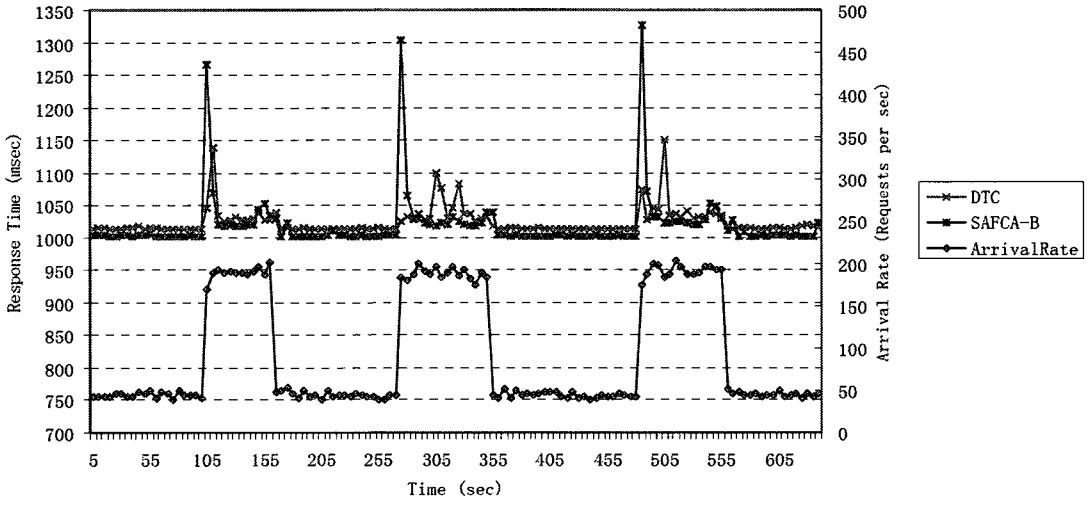

Figure 4.28: Comparison of Response Time for SAFCA-B and DTC

Figure 4.28 demonstrates the response times for SAFCA-B and DTC on the primary y-axis (on the left) and the arrival rate on the secondary y-axis (on the right). Except when the burst first started, SAFCA-B has a better response time than that of DTC.

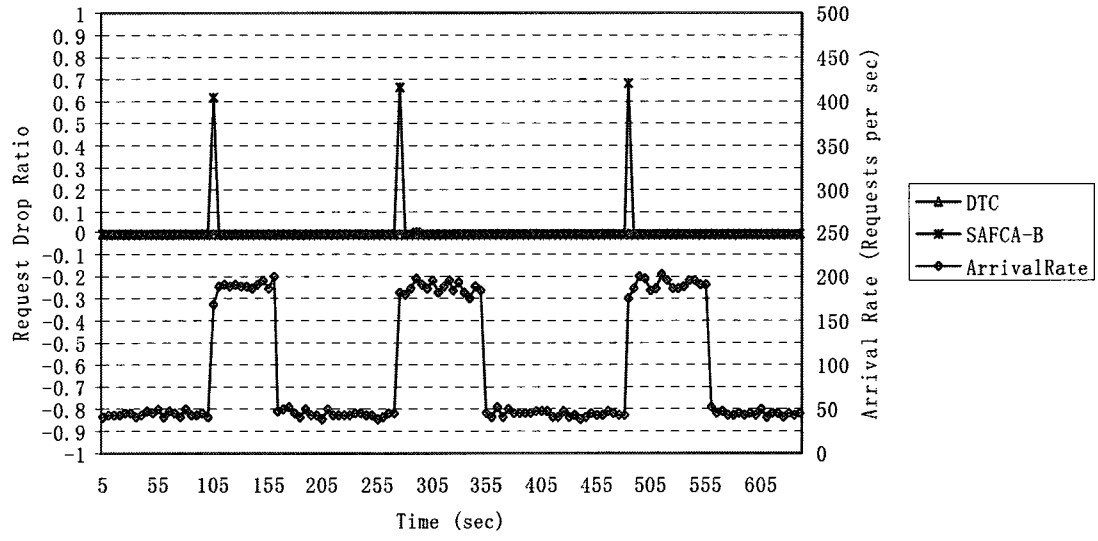

Figure 4.29: Comparison of Drop Ratio for SAFCA-B and DTC

Figure 4.29 demonstrates the drop ratio of for SAFCA-A and DTC on the primary y-axis (on the left) and the arrival rate on the secondary y-axis (on the right). Both SAFCA-B (except when the burst first starts) and DTC have a loss ratio close to 0.

Figure 4.30 illustrates that SAFCA-B has similar CPU utilization as that of DTC in both normal workload and burst workload conditions.

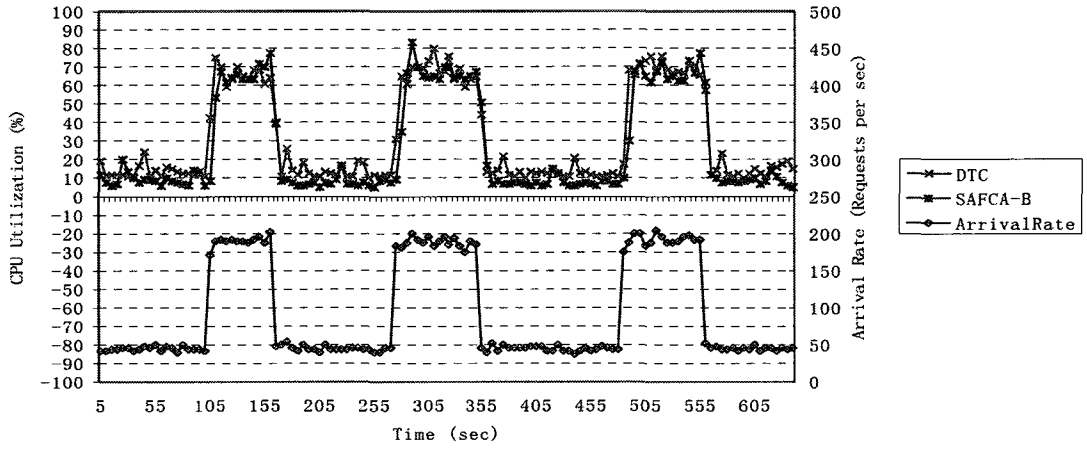

Figure 4.30: Comparison of CPU Utilization for SAFCA-B and DTC

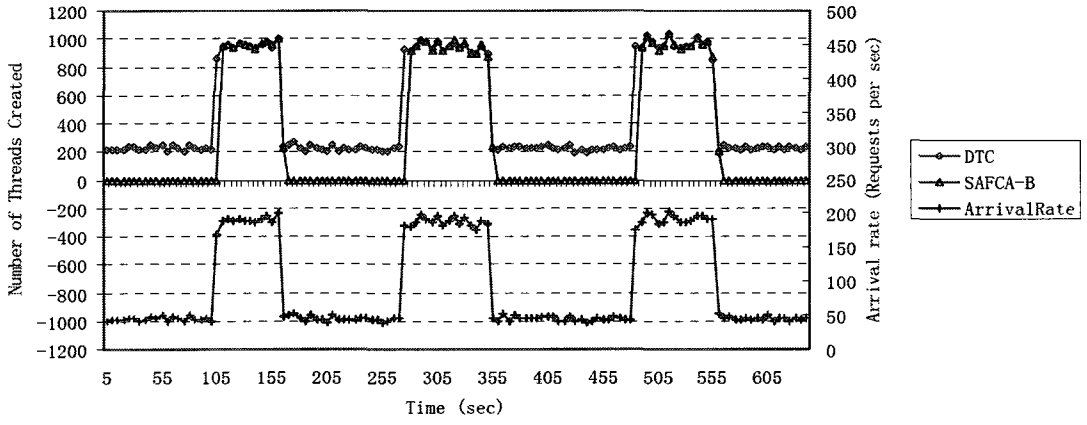

Figure 4.31: Comparison of Number of Threads Created for SAFCA-B and DTC

Figure 4.31 shows that DTC creates more threads and takes up more resources than SAFCA-B during normal workload condition. Each thread created requires memory space, CPU cycles, and the operating system generates the thread creation/destruction overhead. As shown in Figure 4.31, in one sampling interval, DTC creates about 200 new threads under normal conditions. In comparison, instead of creating new threads, only 60 existing threads in the thread pool of SAFCA-B can handle the normal workload. Since normal workload periods are typically much longer than burst periods, SAFCA-B is more resource efficient than DTC. The decision process in SAFCA-A is based on the percentage of change of the arrival rate and the response time, while the decision process in SAFCA-B is based on the standard deviation in the change of the arrival rate and the percentage of change in the response time. The performance results of SAFCA-A and SAFCA-B are very similar.

4.9 Reliability Improvement Using SAFCA-R

Multi-thread or multi-core programming presents a number of challenges since synchronization primitives (such as locks, mutexes, or condition variables) are common sources of bugs. There is a high possibility that concurrency architecture may fail due to deadlock or resource leakage.

In the reliability experiments, the parameters for the HS/HA stand-alone experiment are shown in Table 4.5. When an artificial failure is injected into the system, the system stops until it is rebooted. Restarting from the beginning needs lot of time and the same error is likely to recur for similar scenarios [Whi09]. The *queueSize* value is fixed in this experiment. The experiment in Section 4.6.3 involves varying the *queueSize* value.

Table 4.5: HS/HA Parameters

Experiment Parameters	Value
queueSize	25 requests
threadPoolSize	60

The parameters for the SAFCA-R reliability experiment is shown in Table 4.6. In the SAFCA-R architecture, HS/HA is used initially. When an artificial failure is injected into HS/HA of SAFCA-R, the throughput drops to 0 while the arrival rate remains at the previous level. SAFCA-R detects the error and adapts to another architectural alternative, LFs. The requests in the queue of the original architecture, HS/HA, will be processed first. New arrivals will be put into the queue associated with the second architecture and will be processed subsequently. This is demonstrated in Figure 4.32.

Table 4.6: SAFCA-R Parameters

Experiment Parameters	Value
queueSize(HS/HA)	25 requests
queueSize(LFs)	25 requests
threadPoolSize(HS/HA)	60
threadPoolSize(LFs)	60

As shown Figure 4.32, SAFCA-R can detect the failure and switch to another concurrency architecture during runtime. This is much faster than other rebooting based approaches [Can04]. The recovery time is defined as the sum of the detection time, the

notification time, and the switch over time. The failure detection time of SAFCA-R depends on the sampling interval length and the time the failure occurs, as shown in Figure 4.33.

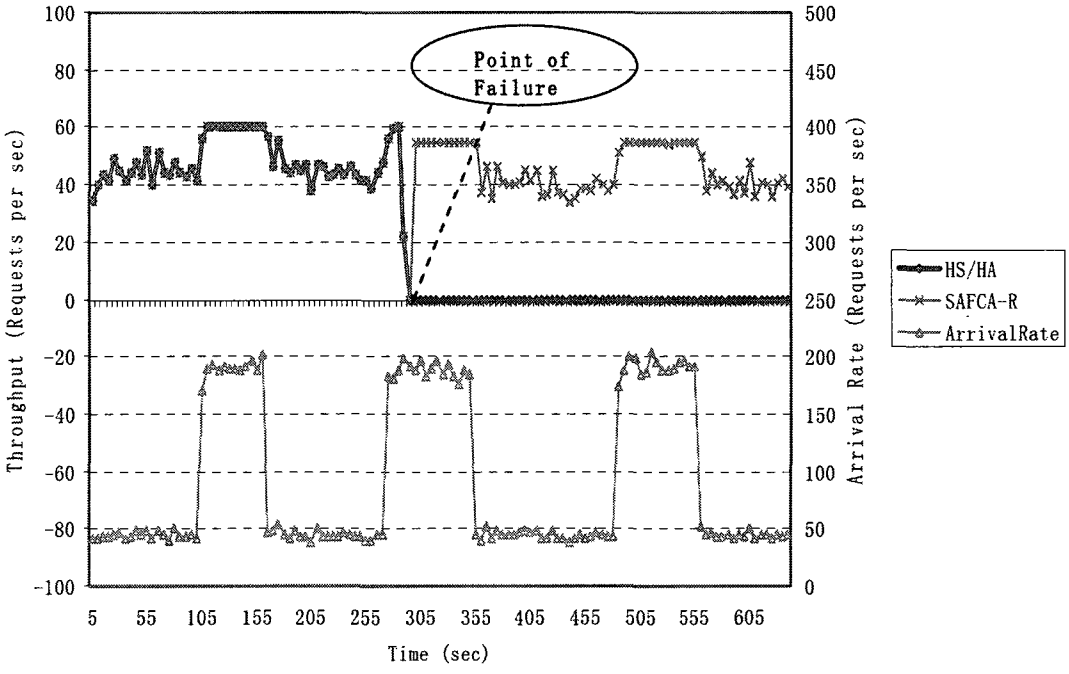

Figure 4.32: Performance Evaluation of SAFCA-R versus HS/HA

The detection time is the time between failure occurrence and the time the system detects failure, as shown in Figure 4.33 (a). As mentioned in Section 3.6.3, SAFCA-R detects failure by checking if the throughput equals to zero, and throughput are sampled at the beginning of each sampling interval. The best detection time is achieved if the failure occurs immediately before the throughput sampling instruction at the beginning of the interval, as illustrated in Figure 4.33 (b). This is because no reply has been sent for

this sampling interval, thus the throughput is zero. Thus, at the end of this interval, the sampled throughput remains zero and failure is detected. The worst case is that the failure occurs immediately after the throughput sampling instruction, as depicted in Figure 4.33 (c).

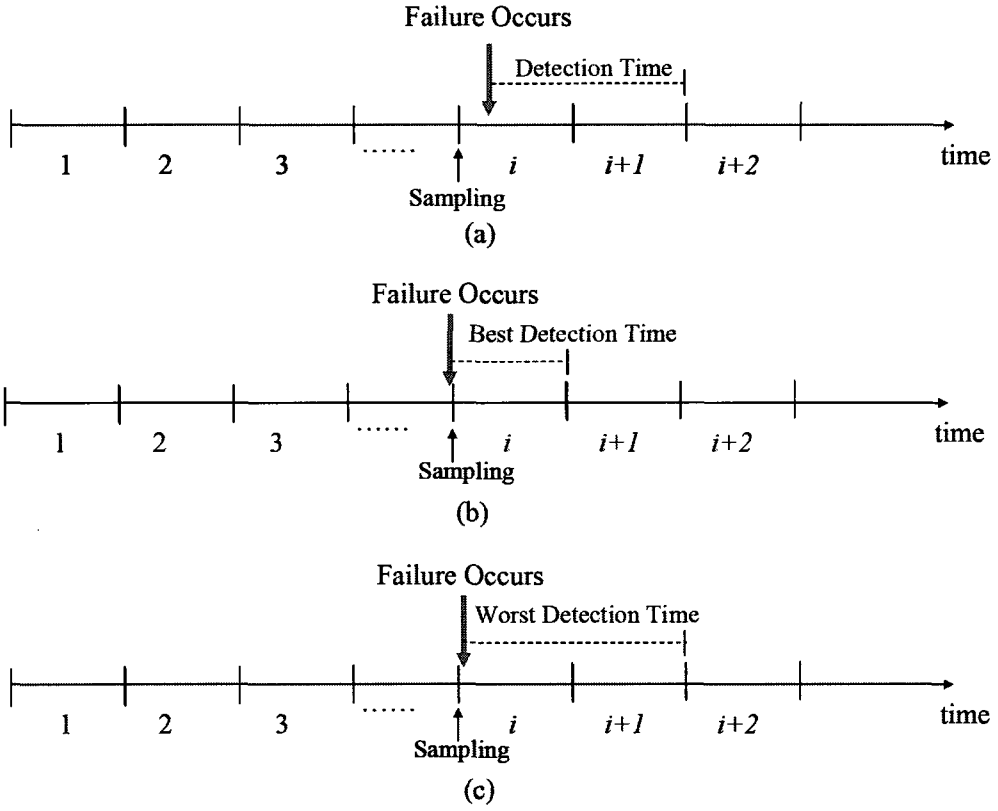

Figure 4.33: Failure Detection Time of SAFCA-R

The throughput for this interval is non-zero in this case, and it will not be until the next interval that the throughput will become zero. For those reasons, the longest detection time for SAFCA-R is two sampling intervals and the shortest detection time is one sampling intervals. Since the sampling interval length for the reliability experiments

is 5 seconds, the worst detection time is 10 seconds and the best detection time is 5 seconds, depending on when the failure is detected.

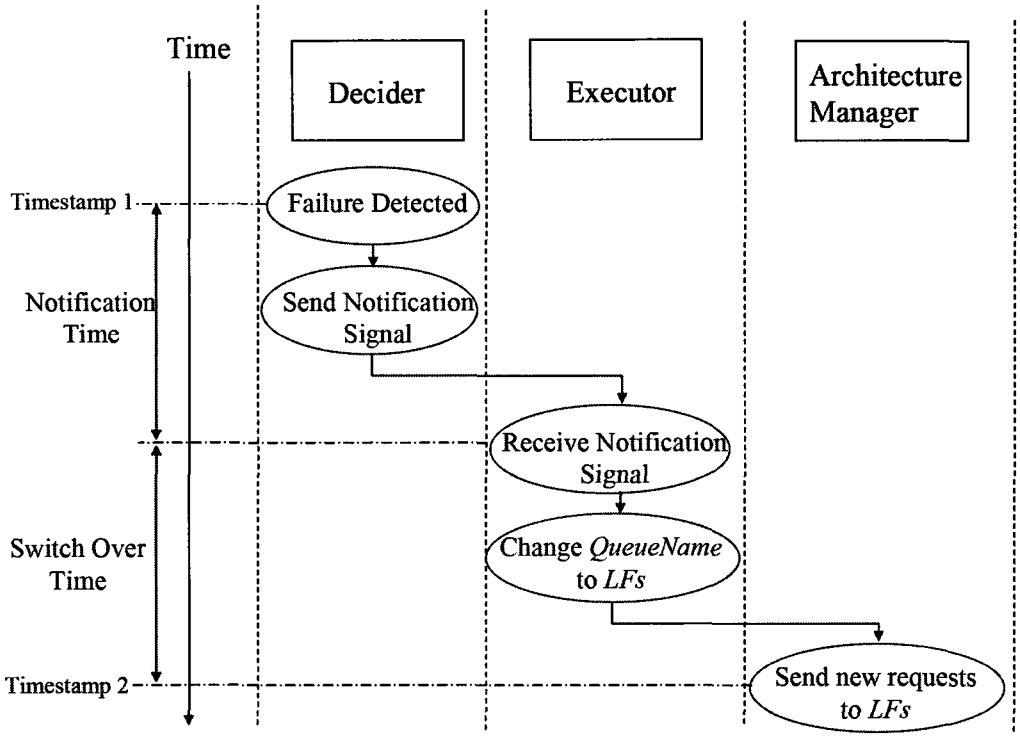

Figure 4.34: Notification Time and Switch Over Time of SAFCA-R

Figure 4.34 shows the notification time and switch over time for SAFCA-R when a failure is detected. When the *Decider* detects a failure, it sends a notification signal to the *Executor*. When the *Executor* receives the signal, it will change *QueueName* from the currently running architecture to the target architecture, e.g., HS/HA to LFs in this experiment. The *Architecture Manager* then begins to send new requests to the queue belonging to LFs architecture. The notification time is the period between the time that the *Decider* detects the failure to the time that the *Executor* receives the signal. The

switch over time is the period from the time the *Executor* receives the signal to the time the *Architecture Manager* begins to send newly arrived requests to the LFs queue.

In order to measure the notification time and switch over time for SAFCA-R, the reliability experiment adds a time stamp when the *Executor* detects the failure and adds another time stamp when the *Architecture Manager* begins to place requests in the LFs queue. The time stamps are measured by importing the java package *java.util.Date* and using the *getTime*() method in the *Date* java class [Oak04]. The resolution of the *getTime()* method is 1 millisecond. Since subtracting the two time stamps yields a result of 0, it can be concluded that the sum of the notification time and the switch over time is less than 1 millisecond. In comparison to the detection time, which is between 5 seconds (one sampling interval) and 10 seconds (2 sampling intervals), the total recovery time is approximately between 5 to 10 seconds.

Without having to reboot the system, SCAFCA-R is able to recover from failure quickly. The recovery time of SCAFCA-R is bounded between one sampling interval length and two sampling interval length. Different policies for failure detection could be adopted and incorporated into the framework depending on the requirements.

4.10 Summary

This chapter has described experiments undertaken to compare the performance of SAFCA-A, SAFCA-B, and SAFCA-Q with stand-alone HS/HA and stand-alone DTC. The results of those experiments are also presented. The reliability analysis of SAFCA-R is also carried out. The performance comparison between LFs and ST and HS/HA is not an emphasis in this thesis because [Zha10] has already presented those results.

As stated previously, SAFCA can include multiple types of concurrency architectures. To handle bursty workloads, the SAFCA design in this work includes two concurrency architectures: HS/HA and DTC. To improve reliability, SAFCA-R includes HS/HA concurrency architecture and LFs concurrency architecture.

The results have shown that SAFCA performs better than HS/HA in terms of response time, request drop ratio, and CPU utilization, while SAFCA performs similarly in comparison with DTC. In terms of response time and number of threads created, SAFCA performs better than DTC under normal workload conditions. As the experiment results have demonstrated, the HS/HA concurrency architecture in SAFCA works well under normal workload conditions. However, if a burst occurs, the size of the thread pool becomes the performance bottleneck. The advantage of HS/HA over DTC is that it does not have to incur thread creation/destruction overhead. The DTC concurrency architecture in SAFCA can create a large number of threads to handle the sudden burst of requests (one thread per request). Through the performance valuation, SAFCA indeed performs better than stand-alone HS/HA under burst load and performs better than stand-alone DTC under non-burst load.

In terms of reliability, a system designed with SAFCA-R can recover from a concurrency architecture failure, where a system running with a single architecture only cannot recover efficiently. Compared with the approach using a redundant standby node, SAFCA-R offers an alternative that is cost effective. SAFCA-R can also recover from failure in a single sampling interval. The thesis focuses on the development of the

framework that supports self-healing. Different policies that consider various parameters, such as sampling interval and switchover time, can be studied and experimented.

Chapter 5 Conclusions and Future Work

This chapter concludes the work presented in this thesis and outlines some potential future research topics.

5.1 Conclusions

In order to increase or maintain the performance, and to effectively utilize resources of multi-tier systems under dynamic workloads, a self-adaptive framework, SAFCA, is developed. In addition, reliability of multi-tier systems can be improved using the framework. In essence, the framework consists of multiple concurrency architectures and an autonomic manager to support adaptation at the architecture level.

According to the results obtained from a number of the experiments, SAFCA improved the performance of a server architecture under different workloads through a self-adaptive mechanism. In comparison to the stand-alone HS/HA architecture, SAFCA exhibited performance gains without the need of over-provisioning. Under normal workload conditions, SAFCA has a better resource usage than HS/HA-only system. On the other hand, in comparison to DTC-only system, SAFCA also demonstrated performance gains under normal workload conditions and exhibited similar performance under bursty workload conditions.

SAFCA also improved the reliability of client-server architecture under single concurrency architecture failure. For a stand-alone architecture, such as HS/HA or LFs, a deadlock or other kinds failure will bring down the system indefinitely. SAFCA-R can recover from a single concurrency architecture failure (for example HS/HA) within a single sampling interval by switching to another architecture (LFs).

SAFCA is implemented using the autonomic computing concept. The decision portion of the feedback control loop is realized using different mechanisms. The self-adaptive decisions in SAFCA-Q are based on the queue length. The decisions in SAFCA-A are based on the percentage of change in arrival rate and the percentage of change in response time. SAFCA-B makes its decision based on change in the arrival rate (in terms of standard deviation) and the percentage of change in response time. SAFCA has also demonstrated improved resource utilization. As stated previously, the performance of these three SAFCA variations were all better than standard alone architectures, while the performance among the SAFCA variations themselves are about the same.

Furthermore, SAFCA-R makes decision based on arrival rate and throughput of the system. SAFCA-R has shown to provide improved reliability than stand-alone concurrency architectures.

5.2 Future Work

Based on the research in this thesis, some future research directions in autonomic control in performance and reliability can be conducted. Those future research directions include:

i. **Autonomic control mechanism:** This thesis adopted a simple control method, as the focus was on the self-adaptive framework. More complex control methods can be incorporated and experimented.

ii. **Performance modeling and evaluation:** Detailed performance modeling and measurements of real systems will be useful in supporting decision-making and autonomic control.

iii. **Reliability:** The current method of detection of failures and deadlocks, in this thesis, is based on the throughput and request-arrival rate: if the arrival rate is non-zero, but the throughput has dropped to zero, a failure is assumed to have occurred. Other efficient mechanisms could be developed to predict failures and/or deadlocks early and start the recovery faster. Different failure detection policies could also be investigated. A policy could even adopt different sampling intervals dynamically depending on the traffic, e.g., a shorter interval could be used for high traffic loads and a longer interval could be adopted for low traffic rates. In addition, failure recognition and switch over time are also areas for further investigations.

References

[Abd02] Abdul-Fatah I. and Majumdar S., "Performance of CORBA-Based Client-Server Architectures", *IEEE Transactions on Parallel and Distributed Systems*, 13(2), Feb. 2002, pp.111-127.

[Ban01] Banks J., Carson J., Nelson B., and Nicol D., Discrete-Event System Simulation, Prentice-Hall, Upper Saddle River, New Jersey, 2001.

[Bec08] Beckmann B., Grabowski L., McKinley P. and Ofria C., "Autonomic Software Development Methodology Based on Darwinian Evolution", *Proc. of the International Conference on Autonomic Computing*, June 2008, pp.203-204.

[Ben09] Ben-Yehuda M., Breitgand D., Factor M., Kolodner H., Kravtsov V. and Pelleg D., "NAP: a Building Block for Remediating Performance Bottlenecks via Black Box Network Analysis", *Proc. of the International Conference on Autonomic Computing*, June 2009, pp.179-187.

[Bru09] Brun Y., Serugendo G. D. M., Gacek C., Giese H., Kienle H., Litoiu M., Muller H, Pezze M. and Shaw M., "Engineering Self-Adaptive Systems through FeedBack Loops", *Software Engineering for Self-Adaptive Systems*, LNCS, 2009, pp.48-70.

[Can04] Candea G., Kawamoto S., Fujiki Y., Friedman G. and Fox A., "Microreboot – A Technique for Cheap Recovery", *Proc. of the 6th Symposium on Operating Systems Design and Implementation*, December 2004, pp.31-44.

[Che09] Cheng S., poladian V., Garlan D. and Schmerl B., "Improving Architecture-Based Self-Adaptation through Resource Prediction. Self-Adaptive Systems", *Software Engineering for Self-Adaptive Systems*, LNCS, 2009, pp.71-88.

[Dob06] Dobson S., Denazis S., Fernandez A., Gaiti D., Gelenbe E., Massacci F., Nixon F., Saffre F., Schmidit N. and Zambonelli F., "A survey of autonomic communications." *ACM Transactions Autonomous Adaptive Systems,* 1(2), 2006, pp.223-259.

[EMA06] Enterprise Managment Associates, *Practical Autonomic Computing: Roadmap to Self-Managing Technology*. Tech. Rep., IBM, Boulder, CO, Jan 2006.

[Fra96] Franks G., Majumdar S., Neilson J., Petriu D.C., Rolia J., and Woodside C.M.:, "Performance analysis of distributed server systems", *Proc. of the 6th International Conference on Software Quality*, 1996, pp.15-26.

[Geo02] Georgiadis I., Magee J., and Kramer J., "Self-Organising Software Architectures for Distributed Systems", *Proc. of the 1st Workshop on Self-Healing Systems*, 2002, pp.33-38.

[Gra09] Grandis P. and Valetto P., *Elicitation and Utilization of Utility Functions for the Self-assessment of Autonomic Applications*, Tech. Rep., Drexel University, Department of Computer Science, Jan 2006.

[Hag09] Hagimont D., Stolf P., Broto L., and Palma N.: Component-Based Autonomic Management for Legacy Software, *Autonomic Computing and Networking*, Springer US, 2009, pp.85-104.

[Har05] Harold E., *Java network Programming*, O'Reilly, Sebastopol, CA, USA, 2005.

[Hyd99] Hyde P., *Java Thread Programming*, Sams, Indianapolis, In, USA, 1999.

[IBM06] IBM Autonomic Computing Architecture Team, *An Architectural Blueprint for Autonomic Computing*, Tech. Rep., IBM, Hawthorne, NY, June 2006.

[Jen08] Jenkov J. "Thread Signalling", Jenkov ApS, 2008, http://tutorials.jenkov.com /java-concurrency/thread-signaling.html#sharedobjects, last accessed on Jan. 21st, 2010.

[Kep03] Kephart J. and Chess D., "The Vision of Autonomic Computing", *Computer*, 36 (1), Jan 2003, pp.41–52.

[Koc04] Kocher D., "Concurrency Patterns", *Seminar Software Architecture, Institute of Information University of Zurich*, Nov 2004., http://www.ifi.uzh.ch/swe/teaching /courses/seminar2004/abgaben/Kocher_ConcurrencyPatterns.pdf, last accessed on Jan. 21st, 2010.

[Lea99] Lea D., *Concurrent Programming in Java: Design Principles and Patterns*, Addison-Wesley, Boston, MA, USA, 1999.

[Liu08] Liu Y., Tan M., Gorton I. and Clayphan A., "An Autonomic Middleware Solution for Coordinating Multiple QoS Controls", *Proc. of the 6th International Conference on Service-Oriented Computing*, 2008, pp.225-240.

[Lun06] Lung C.-H., et al., "Architecture-Centric Software Generation: An Experimental Study on Distributed Systems", *Generative Programming and Component Engineering for QoS Provisioning in Distributed Systems, in collaboration with 5th International Conference on Generative Programming and Component Engineering*, Oct 2006.

[Lun07] Lung C.-H., Zhang X., Franks G., Zaman M., "Towards A Performance-Oriented Self-Adaptive System", *Proc. of the 6th International Workshop on System and Software Architectures,* June 2007.

[Lun09] Lung C.-H. and Zhang X., "Treating Software Design as a Service to Manage Complexity", *Proc. of IEEE International Conference on Service-Oriented Computing and Applications,* Dec 2009.

[Maj03] Majumdar S., Shen E. and Abdul-Fatah I., "Performance of adaptive CORBA middleware", *Journal of Parallel and Distributed Computing*, 2004, pp.201-218.

[Mar99] Martin F., Beck K., Brant J., Opdyke W., and Roberts D. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Boston, MA, USA, 1999

[Mi09] Mi N., Cascale G., Zhang Q., Riska A., and Smirini E., "Autocorrelation-Driven Load Control in Distributed Systems", *Proc. of 17th Annual Meeting of the IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, Sept 2009, (to appear).

[Ms06] Microsoft Support, "How to manage System Monitor counters in Windows XP", Microsoft, June, 2006, http://support.microsoft.com/kb/305610, last accessed on Jan. 21st, 2010.

[Mul06] Muller H. A., O'Brien L., Klein M., and Wood B., *Autonomic Computing*, Tech. Rep., Carnegie Mellon University, Software Engineering Institute, April 2006.

[Oak04] Oaks A. and Wong H., *Java Threads, 3rd Edition*, O'Reilly, Sebastopol, CA, USA, 2004.

[Oso07] Osogami T. and Kato S., "Optimizing System Configurations Quickly by Guessing at the Performance", *Proc. of 2007 ACM Special Interest Group on Measurement and Evaluation*, June 07, pp.145-156.

[Par07] Parashar M., Hariri S.: *Autonomic Computing*, CRC Press, New York, NY, USA, 2007.

[Reh09] Rehak M., Staab E., Fusenig V., Stiborek J., Grill M., Bartos K., Pechoucek M. and Engel T., "Threat-Model-Driven Runtime Adaptation and Evaluation of Intrusion Detection System", *Proc. of the International Conference on Autonomic Computing*, June 2009, pp.65-66.

[Sch95] Schmidt D. and Cranor C., "Half-Sync/Half-Async: A Pattern for Efficient and Well-structured Concurrent I/O", *Proc. of the 2nd Pattern Languages of Programs Conference Monticello*, Sept 1995.

[Sch00a] Schmidt, D., Stal M., Rohnert, H., and Buschmann, F., *Pattern- Oriented Software Architecture, Volume 2: Patterns for Concurrent and Networked Objects*, John Wiley & Sons, Hoboken, New Jersey, USA, 2000.

[Sch00b] Schmidt D., O'Ryan C., Pyarali I., Kircher M., and Buschmann F., "Leader/Followers: A Design Pattern for Efficient Multi-threaded Event Demultiplexing and Dispatching," *Proc. of the 7th Pattern Languages of Programs Conference*, Aug 2000.

[Sch07] Schmerken I., "Can the Market's Systems Keep Up With Electronic Trading?", *Wall Street & Technology,* Feb 2007, pp.20-23.

[Shi00] Shirazi J., *Java Performance Tuning*, O'Reilly, Sebastopol, CA, USA, 2000.

[Shi06] Shin M. and Paniagua F., "Self-Management of COTS Component-Based Systems Using Wrappers", *Proc. of 30th Annual International Computer Software and Applications Conference,* 2006, pp.33-36.

[Urg05] Urgaonkar, B. and Chandra, A. "A smart hill-climbing algorithm for application server configuration", *Proc. of the Second International Conference on Automatic Computing*, 2005.

[Wel00] Welsh M., Gribble S., Brewer E., and Culler D., *A Design Framework for Highly Concurrent Systems*, Tech. Rep., UC Berkeley, April 2000

[Wel02] Welsh M., and Culler D., "Overload Management as a Fundamental Service Design Primitive", *Proc. of the 10th ACM Special Interest Group on Operating Systems European Workshop,* Sept 2002, pp.63-69.

[Wel03] Welsh M., and Culler D., "Adaptive Overload Control for Busy Internet Servers", *Proc. of the 4th USENIX Conference on Internet Technologies and Systems*, March 2003, pp. 4.

[Whi09] White J., Doughtery B., Strowd H., and Schmidt D., "Creating Self-healing Service Compositions with Feature Models and Microrebooting", *International Journal of Business Process Integration and Management*, 4(1), 2009, pp. 35 - 46.

[Zha10] Zhang X., Lung C.-H., and Franks G., "Towards Architecture-based Autonomic Software Performance Engineering", to appear in the *Proc. Of the 4th Conference on Software Architectures*, March 2010.