**TABLE OF CONTENTS**

**CHAPTER NO. TITLE PAGE NO.**

**ABSTRACT** i

**LIST OF FIGURES**  **ii**

1. **INTRODUCTION**

* 1. **SYSTEM REQUIREMENTS**

1.1.1. HARDWARE REQUIREMENTS .

1.1.2. SOFTWARE REQUIREMENTS

**1.2 PROBLEM ANALYSIS**

**1.3 FEASIBILITY STUDY**

**1.4 IMPLEMENTATION PLAN**

**1.5 PHASE DESCRIPTION**

**1.6 MODULE DESCRIPTION**

**APPENDIX 1 SCREENSHOTS**

**APPENDIX 2 SAMPLE CODINGS**

**2. REFERENCES**

**LIST OF FIGURES**

**CHAPTER NO. TITLE PAGE NO.**

1. **SYSTEM ARCHITECTURE DIAGRAM**
2. **CLASS DIAGRAM**

**Abstract:**

With an increasing of the traffic on Internet, a Web-server usually overloads during a heavy-request period. Web-server cluster architectures have been widely employed to provide high-efficiency Internet services. This paper is concerned with load balancing control of Web-server clusters in a scalable server model. A Web server is modeled as a tank with an inflow and an outflow, and hence a Web-server cluster is modeled as a multiple-tanks system with certain inflow constraint using multi threaded and control technique. We investigate how to minimize the resource utilization of servers in the worst case, aiming at improving load balancing among clustered servers. Our contribution is twofold. WEB applications make it possible to deliver critical services provided by organizations directly to clients we propose an enhanced application placement framework, which complements previous works and has contributions in the following four aspects.

A novel framework based on binary search is proposed to detect an optimal load balancing solution. Second, we define system cost as the weighted combination of both placement change and inter-application communication cost. Extensive experiments have been conducted and effectively demonstrate that: 1) the proposed framework achieves a good allocation for clustered web applications. In other words, requests are evenly allocated among servers, and throughput is still maximized; 2) the total system cost maintains at a low level; 3) our algorithm has the capacity of approximating an optimal solution within polynomial time and is promising for practical implementation in real deployments. We have presented a novel framework and a practical algorithm for application placement motivated by the desire to minimize worst case server utilization and improve load balancing.

**Aim and Scope of the Project:**

The current demand for high performance computing is pushing the hardware and software to new limits. As Internet services become more popular and pervasive. More and more content is added online everyday and user s request for information is growing exponentially. A critical problem that arises is managing the performance of services under extreme overload. Building highly concurrent systems, such as large-scale Internet services, requires managing many information flows at once and maintaining peak throughput when demand exceeds resource availability.The real problem is scalability of services.Serve a single user as well as it can serve hundreds or thousands of simultaneous users. The designs of existing operating systems are primarily derived with multiprogramming: allowing multiple applications, each with distinct resource demands, to safely and efficiently share a single set of resources. Internet services have become a vital resource for many people. For example, the Internet based services like e-mail, stock trading and the business web sites are often considered as indispensable. Internet services are a relatively new application domain, which presents unique challenges for OS design. In contrast to the batch processing and interactive workloads for which existing operating systems have been designed, Internet services support a large number of concurrent operations and exhibit enormous variations in load. The number of concurrent sessions and hits per day to Internet sites translates into an even higher number of I/O and network requests, placing great demands on underlying resources. Microsoft’s websites receive over 300 million hits with 4.1 million users a day; Lycos has over 82 million page views and more than a million users daily. Yahoo has over 900 million page views daily. AOL’s Web caches service over 10 billion hits a day.

There are four main reasons that existing OS designs fail to mesh well with the needs of Internet services:

* Inefficient concurrency mechanisms
* Lack of scalable I/O interfaces
* Transparent resource management

**Overview of the Project:**

In order to ensure that the quality of service (QoS) perceived by end clients is acceptable, the servers must include techniques and mechanisms that guarantee a minimum level of QoS. Although QoS has multiple aspects such as response time, throughput, availability, reliability, and security, the primary aspect of QoS considered in this work is related to response time. One commonly adopted structure of Web-server clusters is illustrated in the QoS Mechanism. The project mainly refers to the Clientas the software entity that acts as the Web-user agent. It issues the user requests and transmits them to the Web-server; it receives the request results from the Web-server and presents them to the Web-user. The schematic diagram of this class of Web server clusters can be simplified. Where the dispatcher receives user requests and then dispatches them among the *N* Web-servers. Traffic balancing among servers of Web-server cluster is needed to improve the performance of a Web-server system. Processes and threads are well-supported models of concurrent programming. Threading allows programmers to write straight-line code and rely on the operating system to overlap computation and I/O by transparently switching across threads. Transparent resource virtualization prevents applications from making informed decisions, which are vital to manage excessive load. Much work has focused on performance and robustness for specific services. However, with services becoming increasingly dynamic and flexible, this engineering burden becomes excessive.

**Existing System:**

Most event-based systems have one execution stream and require no true concurrency. The event monitor loops infinitely waiting for an event to happen and to pass it to the handler. In the case of long non-uniform handlers running time, when a handler takes a longer time to execute, the others are blocked and the application may become non-responsive.



**Fig 1.1** Staged Event Driven Architecture

However, as there is no state maintained between events, this makes implementation much harder and the internal algorithm becomes hard to understand, and in some cases very complicated. Event systems hide the control flow through an application, making it difficult to understand the cause and affect relationships when examining source code and when debugging. For instance, many event systems invoke a method in another module by sending a “call” and then waiting for a “return” in response. Event driven server designs can often be very complex, requiring application-specific request scheduling.

**PROPOSED SYSTEM**

**THREADS**

In traditional operating systems, each process has an address space and a single thread of control. There are frequently situations in which it is desirable to have multiple threads of control in the same address space running in quasi-parallel, as though they were separate processes. Threads are generally more powerful than events and closer to the parallel applications.

**Table 1.1:** Thread Vs Events

|  |  |  |
| --- | --- | --- |
| **S.No.** | **Events** | **Threads** |
| 01. | Event handlers | Monitors |
| 02. | Events accepted by a handler | Functions exported by a module |
| 03. | SendMessage/AwaitReply | Procedure call, or fork/join |
| 04. | Send Reply | Return from procedure |
| 05. | Waiting for messages | Waiting on condition variables |

The concurrency in modern server application generally starts from independent requests, and similar pieces of code handle them in parallel. The algorithm behind an application and data flow is easier to understand as the programmer does not need care about transferring states between different parts of the application, e.g., between different methods implemented in separated modules, that implement successive operations. This way a threaded application can have long-lived stateful handlers without blocking.

**Thread Usage**

The main reason for having threads is that in many applications, multiple activities are going on at once. Some of these may block from time to time. By decomposing such an application into multiple sequential threads that run in quasi-parallel, the programming model becomes simpler.

Instead of thinking about interrupts, timers, and context switches, we can think about parallel processes. Only now with threads we add a new element: the ability for the parallel entities to share an address space and all of its data among themselves.

A second argument for having threads is that since they do not have any resources attached to them, they are easier to create and destroy than processes. In many systems, creating a thread goes 100 times faster than creating a process. When the number of threads needed changes dynamically and rapidly, this property is useful.

A third reason for having threads is also a performance argument. Threads yield no performance gain when all of them are CPU bound, but when there is substantial computing and also substantial I/O, having threads allows these activities to overlap, thus speeding up the application.

Finally, threads are useful on systems with multiple CPUs, where real parallelism is possible.

**Threaded Servers**

A simple threaded implementation of this server uses a single, dedicated thread to service the network, and hands off incoming tasks to individual task handling threads, which step through all of the stages of processing that task. An optimization of this simple scheme creates a pool of several threads and dispatches tasks to threads, thereby amortizing the high cost of thread creation and destruction.

In steady state, the number of threads T that execute concurrently in the server is SxL. As the per-task latency increases, there is a corresponding increase in the number of concurrent threads needed to absorb this latency while maintaining a fixed throughput, and likewise the number of threads scales linearly with throughput for fixed latency.

Threads have become the dominant form of expressing concurrency. Thread support is standardized across most operating systems, and is so well established that it is incorporated in modern languages, such as Java.



**Figure 1.2:** Threaded server: For each task that arrives at the server, a new thread is created to handle the task.

**HARDWARE REQUIREMENTS**

Processor : 733 MHz Pentium III Processor

RAM : 128 MB

Hard Drive : 10GB

Monitor : 14” VGA COLOR MONITOR

Mouse : Logitech Serial Mouse

Disk Space : 1 GB

## SOFTWARE REQUIREMENTS

# Platform : JDK 1.6

Operating System : Microsoft Windows NT 4.0 or

Windows 2000or XP

Program Language : JAVA

# Tool : NETBEANS 6.8

**Literature Review:**

**W**EB applications make it possible to deliver critical services provided by organizations directly to clients. Modern web applications typically run on top of a

middleware system, which is responsible for processing client requests and for allocating resources at a high rate. Clustering technology enables middleware systems to achieve high degrees of scalability and availability. On the other hand,

it also poses great challenges in scalable and high performance computing. For instance, it is often cost-inefficient when designing data centers to simultaneously handle the potential peak demands of all the applications, due to the dynamical

fluctuation of request rate. As a result, the middleware systems are supposed to allow dynamical resource allocation to meet different performance requirements from diverse applications.

The problem becomes *dynamic application placement*: given a set of machines1 with constrained resources2 and a set of Web applications with dynamically changing demands, how many instances of each application should be run, and where should they be placed? In past work, this problem has been generally formulated as a variant of the Class Constrained Multiple-Knapsack Problem, with multiple objectives such as maximizing the throughput of the whole system, and minimizing the disturbance due to application instance placements start/stop, just to name a few. The scheme is advantageous over other placement algorithms in terms of computational scalability, application satisfied demand and placement change. However, its load balancing is not optimized: part of the servers could be on heavy load after an execution of application placement algorithm. We will illustrate this problem by a server load distribution example in Section IV-B: all we need to know here is that when the total system load is 50%, only 22% servers have utilization close to the whole system load; however, there are a number of servers have load higher than 80%, and some of them even have 100% utilization. As a result, the response time on the servers with high utilization could be significantly increased and the clients served by these servers may be exposed to unnecessary long response latency, which is unfavorable for real-time web-based applications such as multimedia streaming. To alleviate this problem, the *worst case* of individual server utilization3 should be minimized and load balancing in the whole system should be improved.

**1.”A Scalable Application Placement Controller for Enterprise Data Centers”**

Chunqiang Tang, Malgorzata Steinder, Michael Spreitzer, and Giovanni Pacifici

Given a set of machines and a set of Web applications with dynamically changing demands, an online application placement controller decides how many instances to run for each application and where to put them, while observing all kinds of resource constraints. This NP hard problem has real usage in commercial middleware products. Existing approximation algorithms for this problem can scale to at most a few hundred machines, and may produce placement solutions that are far from optimal when system resources are tight. In this paper, we propose a new algorithm that can produce within 30 seconds high-quality solutions for hard placement problems with thousands of machines and thousands of applications. This scalability is crucial for dynamic resource provisioning in large-scale enterprise data centers. Our algorithm allows multiple applications to share a single machine, and strives to maximize the total satis\_ed application demand, to minimize the number of application starts and stops, and to balance the load across machines. Compared with existing state-of-the-art algorithms, for systems with 100 machines or less, our algorithm is up to 134 times faster, reduces application starts and stops by up to 97%, and produces placement solutions that satisfy up to 25% more application demands. Our algorithm has been implemented and adopted in a leading commercial middleware product for managing the performance of Web applications.

**2.”Capriccio: Scalable Threads for Internet Services”,** Rob von Behren, Jeremy Condit, Feng Zhou, George C. Necula, and Eric Brewer

This paper presents Capriccio, a scalable thread package for use with high-concurrency servers. While recent work has advocated event-based systems, we believe that thread based systems can provide a simpler programming model that achieves equivalent or superior performance. By implementing Capriccio as a user-level thread package, we have decoupled the thread package implementation from

the underlying operating system. As a result, we can take advantage of cooperative threading, new asynchronous I/O mechanisms, and compiler support. Using this approach, we are able to provide three key features: (1) scalability to 100,000 threads, (2) efficient stack management, and (3) resource-aware scheduling. We introduce linked stack management, which minimizes the amount of wasted stack space by providing safe, small, and non-contiguous stacks that can grow or shrink at run time. A compiler analysis makes our stack implementation efficient and sound. We also present resource-aware scheduling, which allows thread scheduling and admission control to adapt to the system’s current resource usage. This technique uses a blocking graph that is automatically derived from the application to describe the flow of control between blocking points in a cooperative thread package. We have applied our techniques to the Apache 2.0.44 web server, demonstrating that we can achieve high performance and scalability despite using a simple threaded programming model.

**3.”Cooperative Task Management without Manual Stack Management** or, **Event-driven Programming is Not the Opposite of Threaded Programming”,**

Atul Adya, Jon Howell, Marvin Theimer,William J. Bolosky, John R. Douceur

Cooperative task management can provide program architects with ease of reasoning about concurrency issues. This property is often espoused by those who

recommend “event-driven” programming over “multithreaded” programming. Those terms conflate several issues. In this paper, we clarify the issues, and show how one can get the best of both worlds: reason more simply about concurrency in the way “event-driven” advocates recommend, while preserving the readability and maintainability of code associated with “multithreaded” programming. We identify the source of confusion about the two programming styles as a conflation of two concepts: *task management* and *stack management*. Those two concerns define a two-axis space in which “multithreaded” and “event-driven” programming are diagonally opposite; there is a third “sweet spot” in the space that combines the advantages of both programming styles. We point out pitfalls in both alternative forms of stack management, *manual* and *automatic*, and we supply techniques that mitigate the danger in the automatic case. Finally, we exhibit adaptors that enable automatic stack management code and manual stack management code to interoperate in the same code base.

**4.”Improving the Fault Resilience of Overlay Multicast for Media Streaming”,** Guang Tan, Stephen A. Jarvis and Daniel P. Spooner

This paper addresses the problem of fault resilience of overlay-based live media streaming from two aspects: (1) how to construct a stable multicast tree that minimizes the negative impact of frequent member departures on existing overlay, and (2) how to efficiently recover from packet errors caused by end-system or network failures. In particular, this paper makes two contributions: (1) A distributed Reliability-Oriented Switching Tree (ROST) algorithm that minimizes the failure correlation among tree nodes. By exploiting both bandwidth and time properties, the algorithm constructs a more reliable multicast tree than existing algorithms that solely minimize tree depth, while not compromising the quality of the tree in terms of service delay and incurring only a small protocol overhead; (2)

A simple Cooperative Error Recovery (CER) protocol that helps recover from packet errors efficiently. Recognizing that a single recovery source is usually incapable of providing timely delivery of the lost data, the protocol recovers from data outages using the residual bandwidths from multiple sources, which are identified using a minimum-loss correlation algorithm. Extensive simulations are conducted to demonstrate the effectiveness of the proposed schemes.

**Module Description:**

* Request Handler
* Resource Controllers
* Resource Aware Scheduler
* Thread Pool
* Thread pool performance monitor

**1. Request Handler:**

Many server applications, such as Web servers, database servers, file servers, or mail servers, are oriented around processing a large number of short tasks that arrive from some remote source. A request arrives at the server in some manner, which might be through a network protocol (such as HTTP, FTP, or POP), through a JMS queue, or perhaps by polling a database. Regardless of how the request arrives, it is often the case in server applications that the processing of each individual task is short-lived and the number of requests is large.

One simplistic model for building a server application would be to create a new thread each time a request arrives and service the request in the new thread. This approach actually works fine for prototyping. Normally active threads consume system resources.

The thread-per-task approach works quite well with a small number of long-running tasks. The single-background-thread approach works quite well so long as scheduling predictability is not important, as is the case with low-priority background tasks. However, most server applications are oriented around processing large numbers of short-lived tasks or subtasks, and it is desirable to have a mechanism for efficiently processing these tasks with low overhead, as well as some measure of resource management and timing predictability.

**REQUEST HANDLER:**

**URL of Request Page**

Response Scheduler

Client / User Request

**Size of the Page**

**Content Details of Page**

**RH**

**(Request Handler)**

**2. Resource Controllers:**

The controller consists of several components. A *monitor* measures response times for each request passing in to the Thread Pool. Requests are tagged with the current time when they enter the service. At each pool, the request’s response time is calculated as the time it leaves S minus the time it entered the system. While this approach does not measure network effects, we expect that under overload the greatest contributor to perceived request latency will be intra-service response time.

The measured 90th-percentile response time over some interval is passed to the *controller* that adjusts the *admission control parameters* based on the administrator-supplied response-time *target*. In the current design, the controller adjusts the rate at which new requests are admitted into the thread pool or the requests are admitted into the next thread pool.

**FEED BACK CONTROLLER:**

Resource Aware Scheduler

TPPM (Thread Pool Performance Monitor)

Thread Pool Allocation

FBC (Feed Back Controller)

Status Report

1. URL Alive or Not
2. Explorer Details
3. Host Details

**Figure 6. Feedback Controller**

**3. Resource Aware Scheduler:**

A resource handler is one that receives requests from request handler and redirects it to the thread pools to get the job done. But it is not as simple as it looks. The main function of resource handler is, it has to redirect the request, which it got from the resource handler to the thread pool where there is no overloading. This can be achieved through constant listening of thread pool performance monitor. When a request arrives it will look for thread pool performance monitor to analyze the load details for the particular instance. Based on the knowledge gained it will then redirect to the thread pool where the load is minimum. In its worse case if it cannot find any thread pools to be free, the incoming requests will be stored in a buffer memory.

SSA provides application-specific scheduling for thread-based applications. Since SSA uses a cooperative threading model, we can view an application as a sequence of stages, where the stages are separated by blocking points. Our methods are more powerful, however, in that they deduce the stages automatically and have direct knowledge of the resources used by each stage, thus enabling finer-grained dynamic scheduling decisions. In particular, we use this automated scheduling to provide admission control and to improve response time. Our approach allows SSA to provide sophisticated, application-specific scheduling without requiring the programmer to use complex or brittle tuning APIs. Thus, we can improve performance and scalability without compromising the simplicity of the threaded programming model.

Most existing event systems prioritize event handlers statically.. SSA goes one step further by introducing the notion of resource-aware scheduling. In this section, we show how to use resource-aware scheduling that is both transparent and application-specific. Our strategy for resource-aware scheduling has three parts:

1. Keep track of resource utilization levels and decide dynamically if each resource is at its limit.

2. Annotate each node with the resources used on its outgoing edges so we can predict the impact on each resource should we schedule threads from that node.

3. Dynamically prioritize nodes (and thus threads) for scheduling based on Information from the first two parts.

For each resource, we increase utilization until it reaches maximum capacity (so long as we don’t overload another resource), and then we throttle back by scheduling nodes that release that resource. When resource usage is low, we want to preferentially schedule nodes that consume that resource, under the assumption that doing so will increase throughput. More importantly, when a resource is overbooked, we preferentially schedule nodes that release the resource to avoid thrashing. This combination, when used with some hysteresis, tends to keep the system at full throttle without the risk of thrashing. Additionally, resource-aware scheduling provides a natural, workload-sensitive form of admission control, since tasks near completion tend to release resources, whereas new tasks allocate them. This strategy is completely adaptive, in that the scheduler responds to changes resource consumption due to both the type of work being done and offered load.

**RESOURCE SHEDULER:**

Active Client List

Memory Size

Request handler

Page Content Size

URL Size

URL ID

Response Handling the User Request

Group Allocation

Monitor Client (Thread Pool Performance Monitor)

**Figure 5. Resource Scheduler**

**4. Thread Pool:**

* **Risks of using thread pools:**

While the thread pool is a powerful mechanism for structuring multithreaded applications, it is not without risk. Applications built with thread pools are subject to all the same concurrency risks as any other multithreaded application, such as synchronization errors and deadlock, and a few other risks specific to thread pools as well, such as pool-related deadlock, resource thrashing, and thread leakage.

* Deadlock:

With any multithreaded application, there is a risk of deadlock. A set of processes or threads is said to be *deadlocked* when each is waiting for an event that only another process in the set can cause. While deadlock is a risk in any multithreaded program, thread pools introduce another opportunity for deadlock, where all pool threads are executing tasks that are blocked waiting for the results of another task on the queue, but the other task cannot run because there is no unoccupied thread available. This can happen when thread pools are used to implement simulations involving many interacting objects, and the simulated objects can send queries to one another that then execute as queued tasks, and the querying object waits synchronously for the response.

* Resource thrashing:

Threads consume numerous resources, including memory and other system resources. Besides the memory required for the Thread object, each thread requires two execution call stacks, which can be large. In addition, the JVM will likely create a native thread for each Java thread, which will consume additional system resources. Finally, while the scheduling overhead of switching between threads is small, with many threads context switching can become a significant drag on your program's performance.

* Concurrency errors:

Thread pools and other queuing mechanisms rely on the use of wait() and notify() methods, which can be tricky. If coded incorrectly, it is possible for notifications to be lost, resulting in threads remaining in an idle state even though there is work in the queue to be processed.

* Thread leakage:

A significant risk in all kinds of thread pools is thread leakage, which occurs when a thread is removed from the pool to perform a task, but is not returned to the pool when the task completes. One way this happens is when the task throws a Runtime Exception or an Error. If the pool class does not catch these, then the thread will simply exit and the size of the thread pool will be permanently reduced by one. When this happens enough times, the thread pool will eventually be empty, and the system will stall because no threads are available to process tasks.

Tasks that permanently stall, such as those that potentially wait forever for resources that are not guaranteed to become available or for input from users who may have gone home, can also cause the equivalent of thread leakage. If a thread is permanently consumed with such a task, it has effectively been removed from the pool. Such tasks should either be given their own thread or wait only for a limited time.

* Request overload:

It is possible for a server to simply be overwhelmed with requests. In this case, we may not want to queue every incoming request to our work queue, because the tasks queued for execution may consume too many system resources and cause resource starvation. It is up to you to decide what to do in this case; in some situations, you may be able to simply throw the request away, relying on higher-level protocols to retry the request later, or you may want to refuse the request with a response indicating that the server is temporarily busy.

**THREAD POOL:**

Randomized memory Selection

**Thread Pool Memory Size Allocation**

Memory Request

TPPM

Pool manager Lock & Unlock

1. **Thread pool performance monitor:**

It looks after the performance of the thread pools associated with it through a response time controller “T**”**. The response time controller will monitor the load of the Input and Output of its own thread pool and send information to the Thread Pool Performance Monitor whenever there is a fluctuation in performance. The fluctuations in load will be calculated by monitoring the number of requests given as input to the number of requests processed at the output for any unit time. The thread pool performance monitor in-turn sends information to help the resource scheduler in directing the request to the appropriate thread pool for better performance and resource utilization. A separate table will be maintained in the thread pool performance monitor to update the load fluctuations of the thread pool associated with it. This monitoring system will send a copy of the table to the resource scheduler whenever there is an update.

**THREAD POOL PERFORMANCE MONITOR:**

Pool Names

Thread Pool View

Memory Allocation Policy

Indexing

Chaining

Allocation Policy

Status of Memory

Current Size

Memory Sizes

Thread Pool

Thread Pool

…………..

Data Flow Diagram:

If 1 stage free

Req Shift in Process next stage

Req accept in 1st stage if it’s free

Start Server

Set Size in process stage

Process split into no\_ stage

SERVER

No yes

No yes

Req completed in stage1

Shift Process

**Architecture Diagram:**

**SERVER RAM MEMORY** **PROCESSOR**

**REQUEST**

**RESOURCE**

**SCHEDULER**

**THREAD POOL PERFORMANCE MONITOR**

**THREAD POOL**

**T**

**THREAD POOL**

**T**

**T**

**THREAD POOL**

**REQUEST**

**HANDLER**

**Figure 2.** Web Server Cluster Based Scalable Service Architecture

**CHAPTER 3**

**IMPLEMENTATION AND TESTING**

**3.1 IMPLEMENTATION**

**3.1.1 A High-Performance HTTP server**

Web servers form the archetypal component of scalable Internet services. Much prior work has investigated the engineering aspects of building high performance HTTP servers, but we included about load conditioning, robustness and ease of construction. One benefit of studying HTTP servers is that a variety of industry-standards exist to measure their performance.

**3.1.2 Java Threads**

Java is one of a small number of languages that provide support at the language level for the creation and management of threads. Because, threads are managed by the Java Virtual Machine (JVM), not by a user-level library or kernel, it is difficult to classify Java threads as either user or kernel-level. We used Java threads as an alternative to the strict user or kernel level models.

All Java programs comprise atleast a single thread of control. We have a main thread in the JVM and additional threads of control within the program. The typical implementation of the JVM is on top of a host operating system. This setup allows the JVM to hide the implementation details of the underlying operating system and to provide a consistent, abstract environment that allows Java programs to operate on any platform that supports a JVM.

In a traditional thread-per connection web server design, the only overload mechanism generally used is to bound the number of processes (and hence the number of simultaneous connections) that the server will allocate. When all server threads are busy, the server stops accepting new connections; this is type of overload protection. Service response time depends on many factors such as user load; the length of a time a given connection is active, and the type of request (eg. Static versus Dynamic pages).

**3.1.3 Request Handler**

The system has been designed so that multiple requests from multiple clients can be met. A separate list of active client list and their request ID will be maintained. A request is received from the client it will be put down in a queue of a request handler.

**3.1.4 Thread Pools**

This system aims at giving flexibility to the administrator in choosing the size the thread pool based on the traffic needs and available resources. We have a freed hand to choose the thread pool varying from 10KB to 30KB in size. There are several systems, which are used to control the overload. Many of the proposed techniques are based on fixed policies, such as bounding the maximum request rate of requests to some constraint value. When the thread pool is completely loaded, on receiving any more requests from the client, will not be further processed by the Resource Scheduler. Instead it rejects such requests by giving the reply message back to the client. This helps the client to defer/wait for some random period. The selection of incoming packet rates based on an observation of system loads, such as CPU utilization and memory usage.

**3.1.5 Resource Controllers**

A key goal of enabling ease of service engineering is to shield programmers from the complexity of performance tuning. The resource usage is calculated based on observed performance and demand. Abstractly, a controller observes runtime characteristics of the stage and adjusts allocation and scheduling parameters to meet performance targets.

**3.1.6 Resource Scheduler and Thread Pool Performance Monitor**

It aims to redirect the request to the corresponding thread pool where there are unutilized threads. Resource Scheduler has been complemented by Thread Pool Performance Monitor. The TPPM here employs two types of monitoring methodology viz. Indexing and Chaining. It keeps watch of the size of web page requested, its dynamic nature and the way application has been distributed and coordinated.

Based on the parameter listed above, the request will consume processing power ie.threads.The TPPM will send its feedback to the Resource Scheduler for allotment of future request to the Thread Pool where the usage is low. Resource Scheduler will look for the request, check for its possible processing requirements and redirect it to the thread pool where there is enough resources available. When a request has been processed and sent back to the client, the corresponding thread will be relieved. A feedback will be sent to the Resource Scheduler by TPPM to intimate the availability of unused Threads.

**3.2 TESTING**

**3.2.1 Test Setup**

The environment in which SSA is tested with the following system configuration:

**Server Side**

P IV2.8 GHz Processor with dual core

1 GB DDR RAM

120 GB SATA Hard Disk

**Client Side (32 numbers)**

PIV 2.4 GHz Processor

512 MB RAM

80 GB SATA Hard Disk

Client and Server side have been tested with Windows platforms. From the workstations, the requests are generated to the server. The number of requests is additively increased by time.

SSA process controller to test the number of requests from various clients is shown in figure 3.1.

**The Full Placement Algorithm:**

The next problem is how to optimize *p* as quickly as possible. We develop an algorithm based on Binary Search. Instead of a blind probe, the upper bound of *p*, *p*+, and the lower bound of *p*, *p*− are calculated iteratively. In each iteration, either *p*+ or *p*− are updated. The full high level pseudo code is depicted in Algorithm 1 where the function **MaxDemandMinCost** is shown in Algorithm 2 and the function **BoundAcceptable** is shown in Algorithm 3. The general process is composed of three main building blocks: initialization, iterative optimizing and final rebalancing.

**Algorithm 1** PlaceFrame()

**Require:** : output: *L*\_\_

*m*,*n*: the load distribution matrix; *I*\_\_

*m*,*n*: placement matrix.

1: *p*+ = 1, *p*−

= ρ, *p* = 1;

2: **MaxDemandMinChange**(*max*\_*demand*\_ , *s*\_);

3: **while** *p*+−*p*−

*p*+ > ε **do**

4: *p* =

*p*++*p*−

2 ;

5: **MaxDemandMinCost**(*max*\_*demand*\_ , *s*\_, *L*\_

*m*,*n*);

6: **if BoundAcceptable**() **then**

7: *p*+ = *p*; //decrease the upper bound of *p* value

8: **else**

9: *p*−

= *p*; //increase the lower bound of *p* value

10: **end if**

11: **end while**

12: *p*\_

= *p*+;

13: ω\_

*m* =

\_

*n*∈*N L*\_

*m*,*n*;

14: **Final\_Rebalancing**();

15: *I*\_\_

*m*,*n* = *I*\_

*m*,*n*;

**Algorithm 2** MaxDemandMinCost()

**Require:** Input: *p*: the maximal machine utilization threshold.

Output: calculated max satisfied demand *max*\_*demand* and the cost *s*.

*Lm*,*n*: the load distribution matrix.

1: **for** i = 0 to K // K=10 by default; **do**

2: calc\_max\_demand\_satisfied\_by\_current\_placement ();

3: **if** all\_demands\_satisfied **then**

4: **if** worst\_case\_satisfied **then**

5: //the maximal machine utilization

6: //is less than the given threshold *p*;

7: break out of the loop;

8: **end if**

9: **end if**

10: . . . // we omit the details about placement changes.

11: **end for**

*1) Initialization:* In the initialization phase(lines 1-2 in Algorithm 1), we set *p*− with ρ, and *p*+ with 1, *p* = 1. After performing the function**MaxDemandMinChange**() described in [6], we obtain *max*\_*demand*\_ and *c*\_, which will be used as constraint parameters in the later iterative optimizing phase. For simplicity, we omit details about the algorithm in this paper. Briefly speaking, it strives to probe the maximal application demand by means of iteratively making placement changes in order to increase the total satisfied demand, for instance, stopping “unproductive" application instances and

**Algorithm 3** BoundAcceptable()

**Require:** Input: *max*\_*demand*\_ and *max*\_*demand*: the demand satisfied before

and after updating worst case machine utilization constrains; Cost *s*\_ and

*s*: the system cost before and after updating worst case machine utilization

constrains.

1: **if** *max*\_*demand*\_

== *max*\_*demand* **then**

2: **if** BOUNDED **then**

3: **if** *s* > *s*\_ **then**

4: return FALSE;

5: **end if**

6: **end if**

7: return TRUE;

8: **end if**

9: return FALSE;

starting useful ones.

We also made some minor modifications to this function. For example, by integrating an additional optimization objective, the system stops making application changes only when the worst case (the maximal machine utilization) is less than a given threshold *p*.

***2) Iterative Optimizing:***

The system attempts to iteratively decrease the upper bound or increase the lower bound of *p* (lines 3-11 in Algorithm 1) in order to proximate an optimal solution. The revised problem with parameter *p* is then addressed by continuously identifying whether the updated value of *p* is acceptable or not. If the solution is acceptable, *p*+ is updated; otherwise *p*− is updated. As such, the difference between *p*+ and *p*− is decreased. That is, the system strives to find the optimal value for *p* based on binary search. This loop executes until the difference between *p*+ and *p*− is small enough. For an ε−approximation of the optimum value *p*∗, the iterative search can be completed in *O*(*log*2(1/ε)) rounds [20].

The final *p* value is a good approximation to constrain the worst case of the machine utilization across all machines. **MaxDemandMinCost** is an extension of the baseline algorithm to support the minimization of, besides instance start/stop cost, inter-application communication cost. We currently can deal with paired-dependency: two applications have communications with and only with each other; we plan to extend the algorithm to more complex dependency relationships in next step. The main improvement to **MaxDemandMinChange** is the load-shifting part. For example, application *a* and *b* are dependent to each other. **MaxDemandMinCost** algorithm identifies all those servers that already contain both *a* and *b* instances, say, Group 1; and all those servers contain either *a* or *b* instances, say, Group 2. After sorting the machines in an increasing order of residual memory, our algorithm sort those servers in Group 1 and put them in the head of list, and those servers in Group 2 in the tail of list. The intuition here is to shift the load of *dependent application* from Group 2 to Group 1. Instances of *dependent application* in Group 2 servers are more likely to be idle after the load-shifting phase, hence easier to deal with in the placement change phase In the placement change phase, each time we only change one instance per dependent pair. For each pair of *dependent applications*, we find one server in its Group 2 with the most appropriate residual memory and the largest idle CPU power, add the missing application’s instance, and move it to Group 1.

**.**

***3) Final Rebalancing:***

The last step of the algorithm is Final Rebalancing(line 14 in Algorithm 1). In [6], the final load-balancing component from [10] was used, which moves the new application instances across machines to balance the load, while keeping the total satisfied demand and the number of placement changes the same. While the basic idea of our Final Rebalancing component is similar to [6], [10], it differs from previous work in the following ways: it not only keeps the total satisfied demand and the number of placement changes the same, but also keeps the maximal machine utilization less than the given threshold for the worst case. That is, the system attempts to find another load distribution matrix *L*\_\_ that satisfies the same demand for all dynamic clusters while achieving more balanced load across machines. We calculate *L*\_\_ by solving the following optimization problem:

Minimize:

*n*∈*N*

\_\_\_\_\_\_\_

\_

*m*∈*M*

*L*\_\_

*m*,*n*

− ρ ∗ Ω*n*

\_\_\_\_\_\_\_

(4)

Subject to:

\_

*m*∈*M*

*L*\_\_

*m*,*n*

≤ *p*\_ ∗ Ω*n*, ∀*n* ∈ *N*

\_

*n*∈*N*

*L*\_\_

*m*,*n* = ω\_

*m*, ∀*m* ∈ *M*

*I*\_

*m*,*n* = 0 ⇒ *L*\_\_

*m*,*n* = 0, ∀*m* ∈ *M*, ∀*n* ∈ *N*

(5)

Here *p*\_ presents a constraint for the worst case machine utilization and *w*\_

*m* is the satisfied demand of application *m* calculated during the first two phases (line 12-13 in Algorithm 1). The goal of the last phase is to reassign the load across machines, accordingly balancing the load and making all ρ*n* as close to system load ρ as possible. We transform the problem in the last phase into a min-cost flow problem [21] as shown in Fig. 2. In this figure, from left to right, we can see

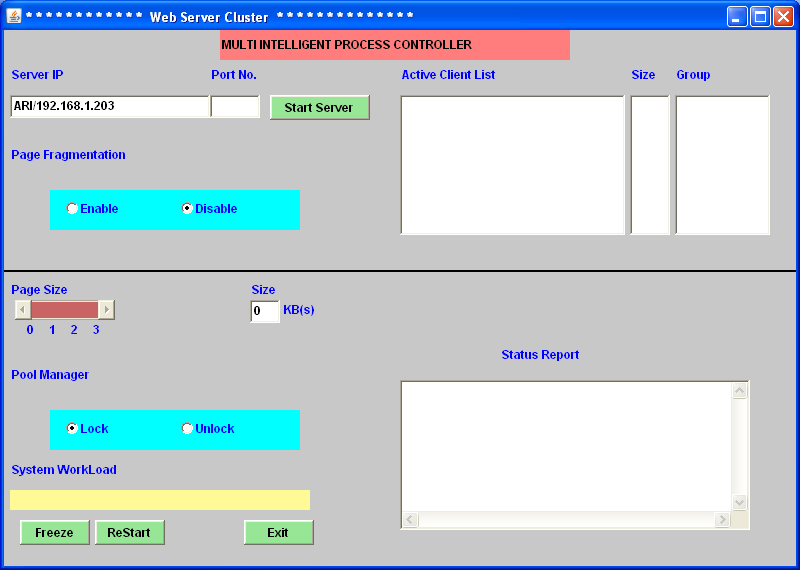
• Source node *S* has outbound edges to every application vertex *m*, where the capacity of the edge *S* → *m* is equal to load-dependent requirement ω\_ *m* and its cost is equal to 0.

• Each application vertex *m* has outbound edges to machine vertices *n* representing the machines that the application is placed on, conform with *Im*,*n*. The capacity of edge *m* → *n* is equal to ω\_ *m* and its cost is equal to 0. • Each machine vertex *n* has an outbound edge to the ideal auxiliary machine vertices *n*\_ that corresponds to the same physical machine. The capacity of the edge *n* → *n*\_ is equal to the desired upper bound usage of the machine ρΩ*n* and its cost is equal to 0. • The rebalancing vertex *R* has inbound edges from machine vertices *n* → *R*, whose capacity is equal to (*p* − ρ)Ω*n*. The cost of these inbound vertices in 1. • The rebalancing vertex *R* has outbound edges to ideal machine vertices *R* → *n*\_. Each such edge has the capacity ρΩ*n* and the cost of 1.

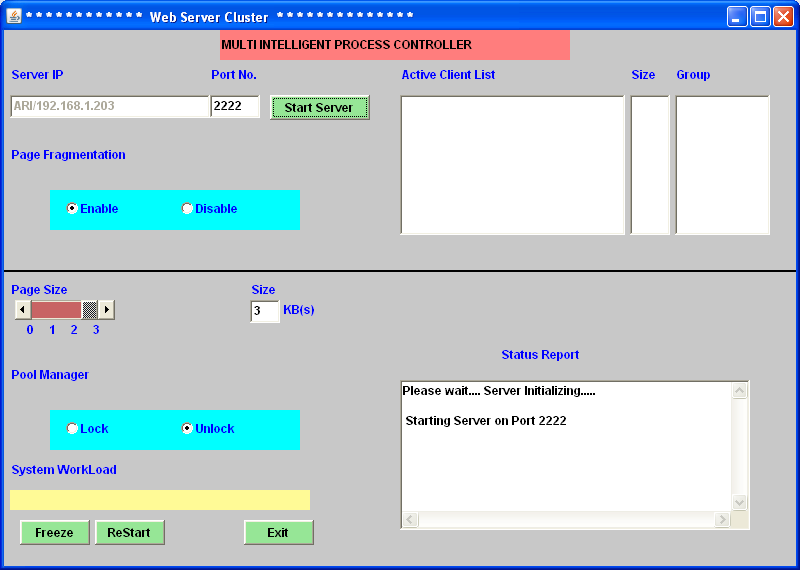
• All ideal machine vertices have an outbound edge to the sink node *T*, with capacity limit equal to ρΩ*n* and the cost of 0. TIAN

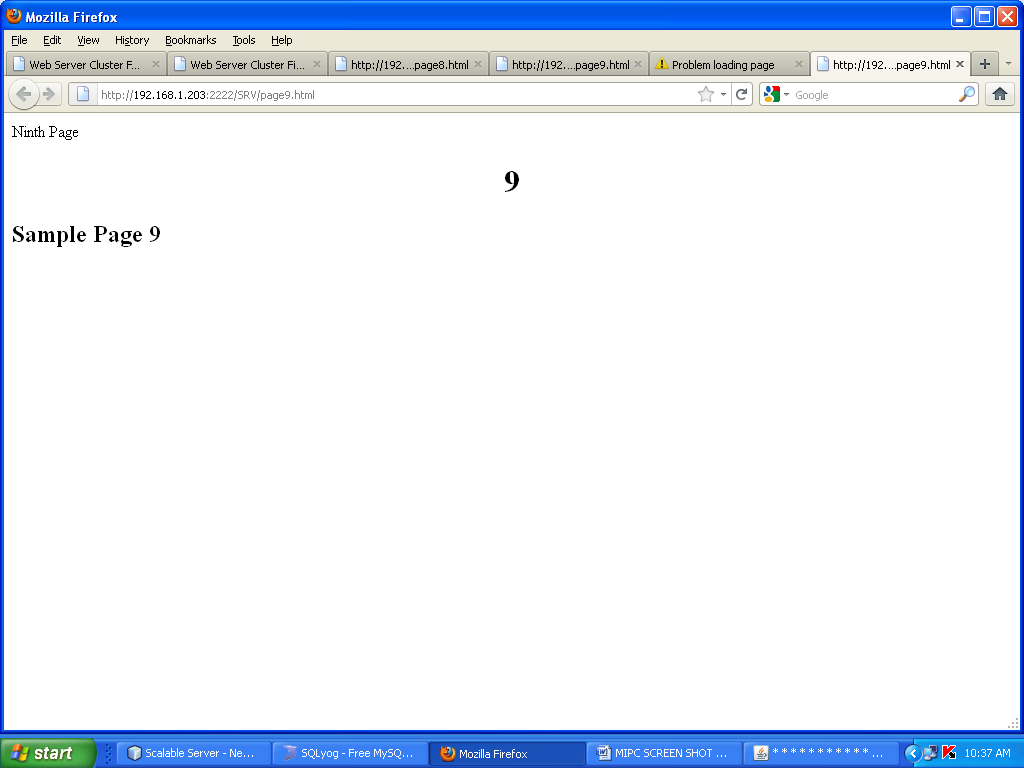
**Screen shots:**

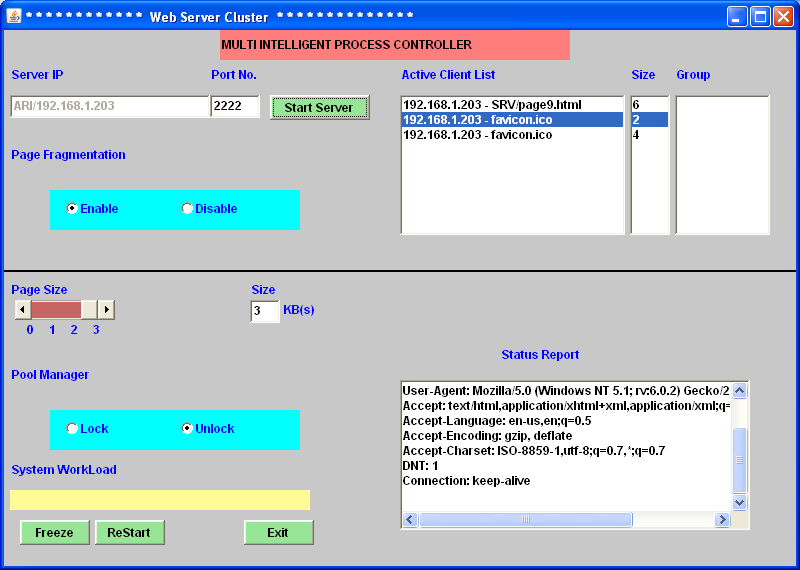
MAIN PAGE :

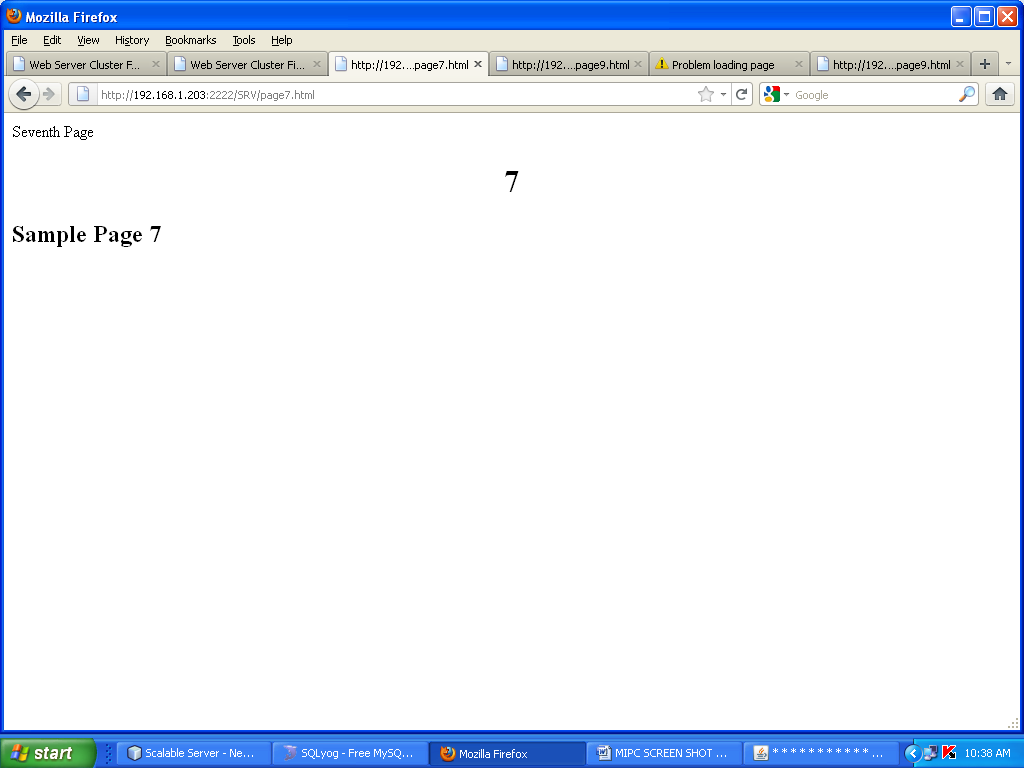


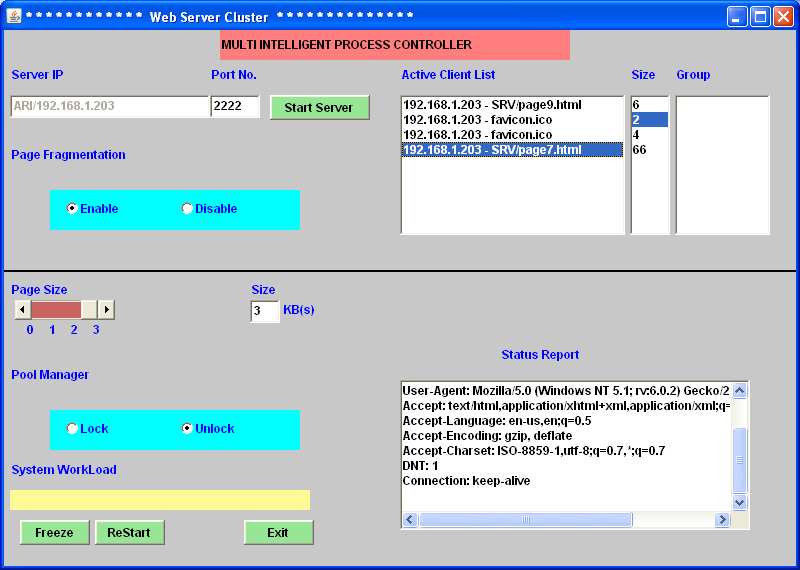
SERVER START WINDOW:

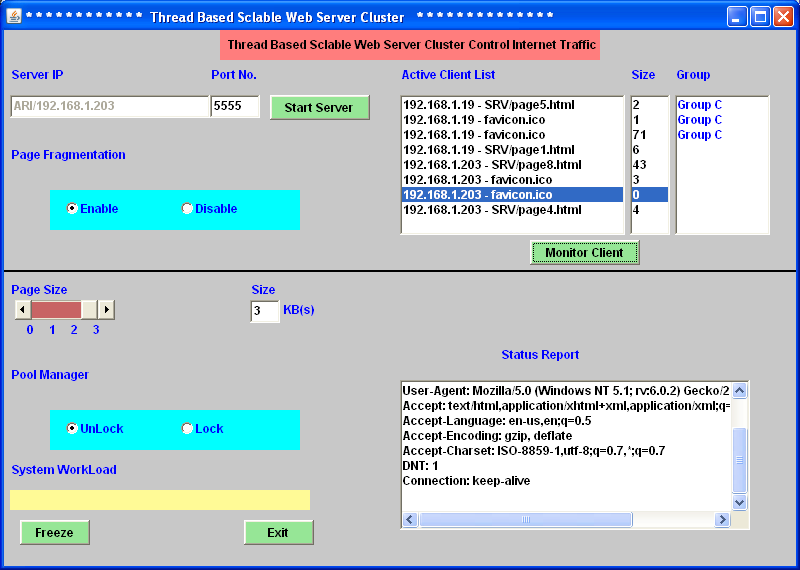


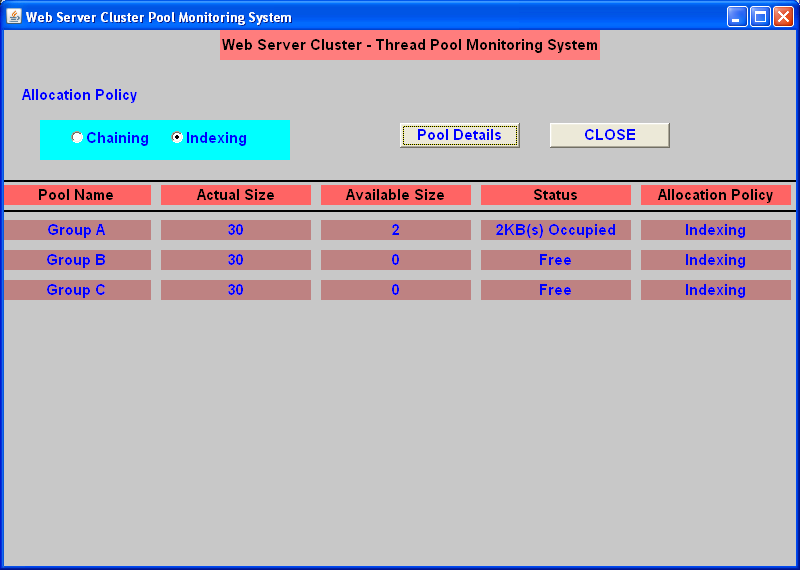


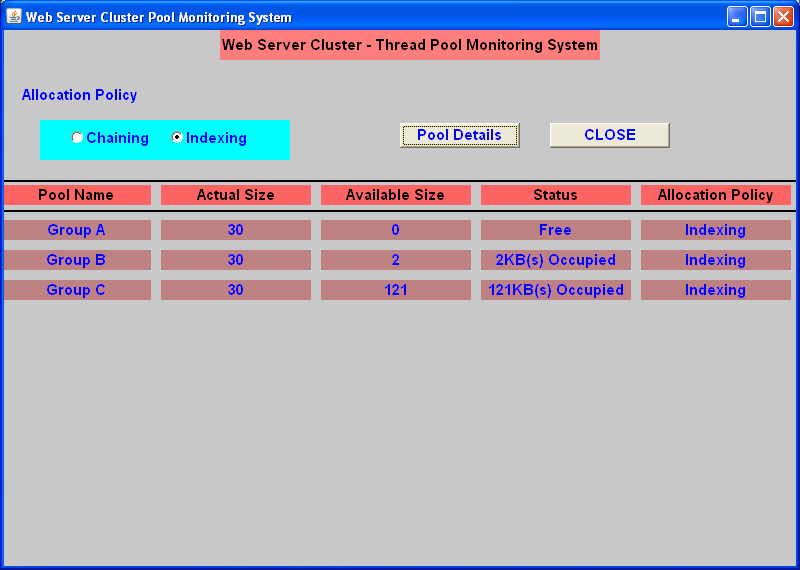


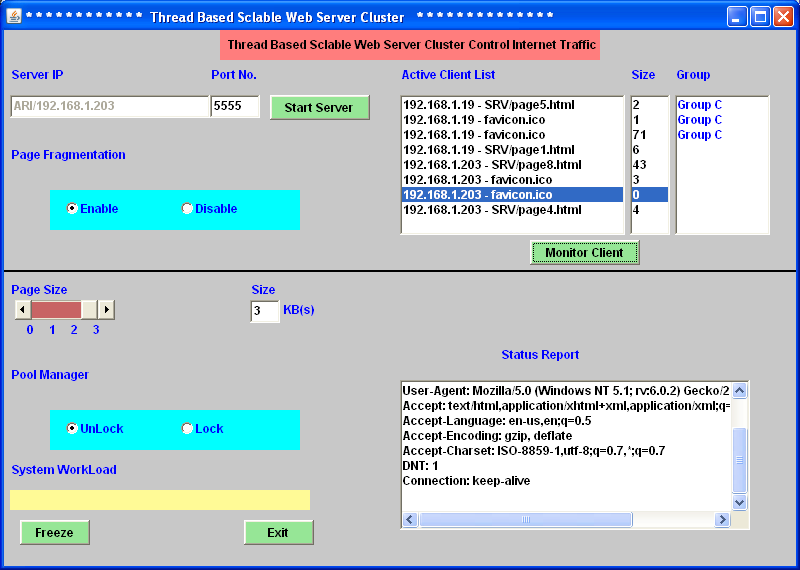


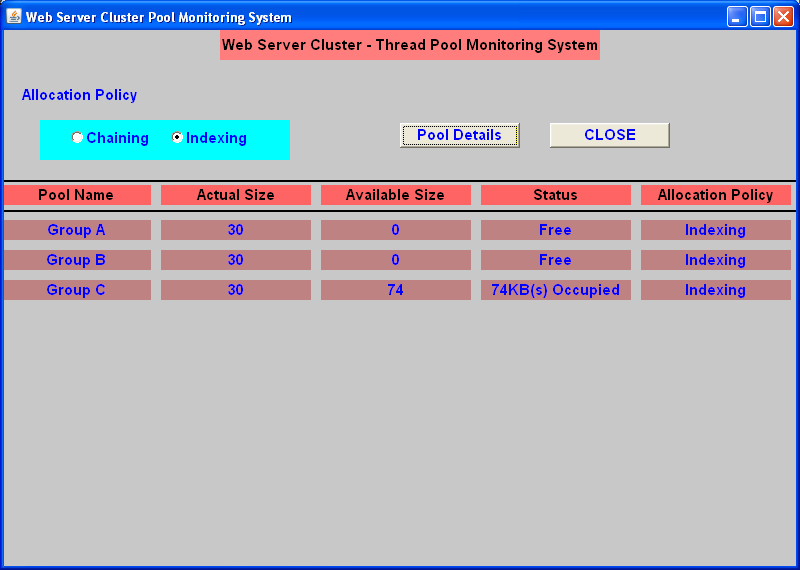












**CONCLUSIONS**

The SSA thread package provides empirical evidence that fixing thread packages is a viable solution to the problem of building scalable, high-concurrent Internet servers. Our experience with writing such programs suggests that the threaded programming model is a more useful abstraction than the event-based model for writing, maintaining, and debugging these servers. By decoupling the thread implementation from the operating system itself, we can take advantage of new I/O mechanisms and compiler support. As a result, we can use techniques such as I/O scalability, load conditioning and resource-aware scheduling, which allow us to achieve significant scalability and performance improvements when compared to existing event-based systems.

**FUTURE WORK**

We are in the process of extending SSA to work with multi-CPU machines. The fundamental challenge provided by multiple CPUs is that we can no longer rely on the cooperative threading model to provide atomicity. We planned to produce the compiler, which can assist the scheduler in making decisions that guarantee atomicity of certain blocks of code at the application level. There are a number of aspects of SSA’s implementation we would like to explore. We believe we could dramatically reduce kernel crossings under heavy network load with a batching interface for asynchronous network I/O. We also expect there are many ways to improve our resource-aware scheduler, such as tracking the variance in the resource usage of blocking graph nodes and improving our detection of thrashing.

Thread packages usually attempt to provide the programmer with the abstraction of an unbounded call stack for each thread. In reality, the stack size is bounded, but the bounds are chosen conservatively so that there is plenty of space for normal program execution. There are several ways in which our stack analysis can be improved.

We can use a conservative approximation of the call graph in the presence of function pointers or other language features that require indirect calls (e.g., higher-order functions, virtual method dispatch, and exceptions). Improvements to this approximation could substantially improve our results. By writing programs in threaded style, programmers provide the compiler with more information about the high-level structure of the tasks that the server must perform.

**REFERENCES:**

[1]D. Li, J. Wu, Y. Cui, and J. Liu, “QoS-aware streaming in overlay multicast considering the selfishness in construction action," in Proc. IEEE INFOCOM, 2007.

[2] G. Tan and S. Jarvis, “Improving the fault resilience of overlay multicast for media streaming," IEEE Trans. Parallel Distributed Syst., vol. 18, no. 6, pp. 721-734, 2007.

[3] A. T. S. Chan, J. Cao, and C. K. Chan, “Webgop: collaborative web services based on graph-oriented programming," IEEE Trans. Syst., Man, Cybernetics, Part A, vol. 35, no. 6, pp. 1874-1885, 2005.

[4] J. Liu, J. Xu, and X. Chu, “Fine-grained scalable video caching for heterogeneous clients," IEEE Trans. Multimedia, vol. 8, no. 5, pp. 1011- 1020, 2006.

[5]K. Shen, H. Tang, T. Yang, and C. L, “Integrated resource management for cluster-based Internet services," in Proc. OSDI, 2002.

[6]C. Tang, M. Steinder, M. Spreitzer, and G. Pacici, “A scalable application placement controller for enterprise data centers," in Proc. International World Wide Web Conf. (WWW), 2007.

[7]H. Shachnai and T. Tamir, “On two class-constrained versions of the multiple knapsack problem," Algorithmica, vol. 29, no. 3, pp. 442-467, 2001.

[8]“Polynomial time approximation schemes for class-constrained packing problems," J. Scheduling, vol. 4, no. 6, pp. 313-338, 2001.

[9] S. Zhou, “A trace-driven simulation study of dynamic load balancing," Univ. California, Berkeley, Tech. Rep. UCB/CSD87/305, 1986.

[10] A. Karve, T. Kimbrel, G. Pacici, M. Spreitzer, M. Steinder, M. Sviridenko, and A. Tantawi, “Dynamic application placement for clustered web applications," in Proc. International World Wide Web Conf. (WWW),2006.