A Content Aware Scheduling System for Network Services in Linux Clusters[1]

Yijun Lu, Hai Jin, Hong Jiang* and Zongfen Han

Huazhong University of Science and Technology, Wuhan, Hubei, 430074, China

yijlu@cse.unl.edu {hjin, zfhan}@hust.edu.cn

*University of Nebraska-Lincoln, Lincoln, NE 68588-0115, USA

jiang@cse.unl.edu

ABASTRACT

With explosive growth of Internet, more and more companies are in need of powerful web servers to support e-commerce and other business activities. To meet this need, cluster architecture has emerged to be the most popular choice for high performance web servers. As one of the most important key technologies, content aware scheduling is becoming a hot research topic. Content-aware scheduling systems have many advantages over other solutions.

In this paper, we design and implement a scheduling system with content awareness for cluster web servers. This system is implemented in Linux kernel. This system is composed of two main modules: the network dispatcher module and the node server module. We also extend this system to support all web services based on TCP, such as support IIOP (Internet Inter-ORB Protocol) and high performance cluster cache server.

The performance of this system is benchmarked with Webbench and httperf. According to the testing results, this system shows good scalability and low response latency.

Keywords

Cluster, Scheduling, Content Awareness, Scalability, Linux, IIOP, Cache Server

INTRODUCTION

A single node or single SMP server hosting a service is no longer sufficient to meet the needs and challenges of companies in the Internet era that require powerful web servers to support e-commerce and other business activities. Cluster-based server has been proven to be an efficient and cost effective alternative to build a scalable, reliable, and high-performance Internet server system. In fact, popular web sites increasingly run Internet services on a cluster of servers (e.g. Alta vista, Netscape, Google), and this trend is likely to accelerate.

As one of the most important technologies in web cluster, *Content Aware Scheduling System* (CASS) is becoming a hot research topic. Content aware scheduling system has many advantages over other solutions: (1) increased performance due to improved hit rates in the node server's main memory caches, (2) increased secondary storage scalability due to the ability to partition the server's database over the different node servers, and (3) the ability to employ node servers specialized for certain types of requests (e.g., audio and video).

Currently, CASS is often implemented with the help of application level programs, such as Apache. For example, Rice University has implemented CASS in kernel level in FreeBSD platform [12], while Harvard University conducted CASS research in the Windows NT platform [14]. The research focus of Rice University is scheduling policies, such as LARD (Locality Aware Request Distribution). Due to the flexibility and popularity of Linux, we implement our content aware scheduling system in Linux. By using a LARD-like scheduling policy, we focus on the content aware scheduling architecture in Linux.

In this paper, we present the main design and implementation issues of this system, including a new, efficient technology used to deliver information with an existing packet in the kernel of Linux. This technology avoids creating a new sk_buff structure to relay necessary information from the dispatcher to a selected node server. In addition, this system can also support other TCP-based network services. More specifically, the support package for IIOP, implemented in this system, demonstrates the system ability to support other network services. The system also facilitates the construction of cluster cache servers.

The rest of this paper is organized as follows. We discuss related work in the next section. Section 3 outlines the design principles of the proposed system. Section 4 describes the overall design of this system and presents the

[1] This paper is supported in part by NSF under grant EPS-0091900

rationale for some design choices. The application of this system is discussed in section 5. Section 6 presents performance evaluation of the system in a web cluster environment. Finally, section 8 concludes the paper with remarks on current and future work.

RELATED WORKS

For cluster computing, the network dispatching technology for clients requests is an important issue. This research issue ranges from the dispatching of common parallel jobs to that of special services such as web services. One of the first techniques emerged is based on the dynamic forwarding according to *Domain Name System* (DNS) [10, 11]. It is argued that a smart client can gain more advantages by allowing the clients to perform load balancing [2]. Other research projects that implemented dispatching in the user space include a network router solution [4], Reverse-proxy [9], SWEB [5] and H-SWEB [3]. These schemes in general require more system resources than kernel level solution. Till now, the most common network service dispatching is based on IP level forwarding [1, 8, 13].

For content aware scheduling, Pai [12] explored the use of content-based request distribution in a cluster web server environment. This work presented an instance of a content-aware request distribution strategy, called LARD (Locality Aware Request Distribution). The strategy achieved both locality, in order to increase hit rates in the web servers memory caches, and load balancing. Performance results of the LARD algorithm showed substantial performance gains over WRR (Weighted Round – Robin).

Zhang [14] explored another content-based request distribution algorithm that looked at static and dynamic content and focused on cache affinity. They conformed the results of Pai by showing that focusing on locality can lead to significant improvements in cluster throughput.

More recently, Mohit Aron [7] presented a new, scalable architecture for content-aware request distribution in web server clusters. Besides supporting content aware scheduling policy, their cluster architecture improved server performance by allowing partitioned secondary storage, specialized server nodes, and request distribution strategies that optimize for locality.

DESIGN PRINCIPLES

In this section, we examine some technical issues that we encountered while designing the prototype system. The first issue concerns with the three-way handshake process to establish a TCP connection. The second one is about packet relaying. The third issue deals with a new efficient technology we developed to deliver information with an existing packet in Linux kernel. The last issue is about the mechanism of generality of content aware scheduling system.

Figure 1 shows the sequence of events in the connection

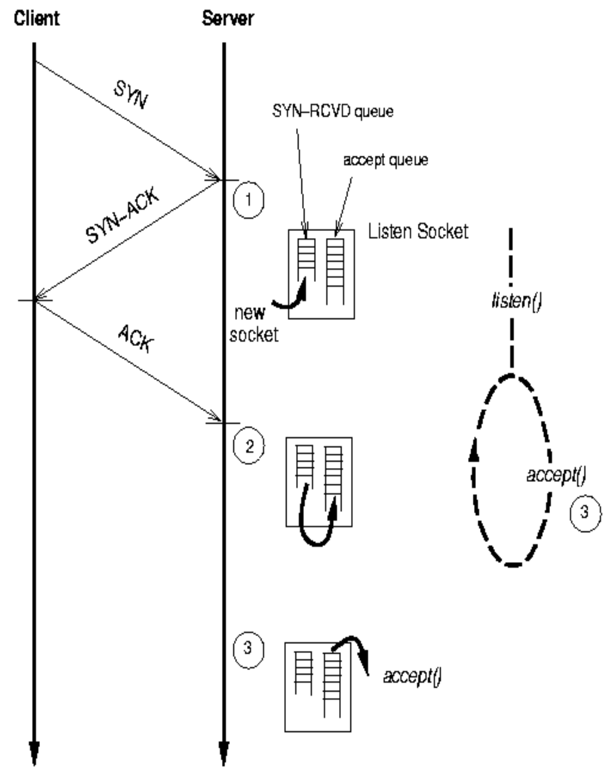

Figure 1 Event Sequence in the Connection Phase of an HTTP Transaction

establishment phase of an HTTP transaction. When starting, a web server process listens for connection requests on a socket bound to a well known port — typically port 80. When a connection establishment request (TCP SYN packet) from a client is received on this socket (Figure 1, position 1), the server TCP responses with a SYN-ACK TCP packet, creates a socket for the new, incomplete connection, and places it in the listen socket's SYN-RECV queue. Later, when the client responds with an ACK packet to the server's SYN-ACK packet (position 2), the server TCP removes the socket created above from the SYN-RECV queue and places it in the listen socket's queue of connections awaiting acceptance (accept queue). Each time the web server process executes the accept() system call (position 3), the first socket in the accept queue of the listen socket is removed and returned. After accepting a connection, the web server - either directly or indirectly - reads the HTTP request from the client, sends back an appropriate response, and closes the connection.

Before receiving the actual request, the server must establish the TCP connection through three-way handshake process with the client. This can cause two main problems in the design of content aware scheduling system. The first problem is that the server does not know which packet will include the client's request at first, it must rely on the TCP states changing rules. To solve this problem, we develop the pseudo-server module to listen and intercept the coming-back TCP packet between the client and the network dispatcher. The pseudo-server's main advantage lies in its avoidance of a lot of single server programs (such as Apache Web Server and CORBA server) installed in the dispatcher. Installing single server programs has two main disadvantages: (1) single server programs consume system resources; (2) it prevents the server program from doing more important jobs, such as configuring Apache for monitor and remote-configuration.

The second problem has to do with how the client's request is relayed to the selected node server. Because the node server has not established the TCP connection through three-way handshake process, the node server's TCP/IP stack won't accept the relayed client's request without any more processing. A common solution is to establish a new socket between the network dispatcher and the selected node server. But this solution will incur more system cost. In this paper, we use the *faking three-way handshake* trick to reduce system cost.

The third issue concerns with the information delivery mechanism in Linux kernel. In order to fake the three-way handshake process, the node server requires the necessary three-way handshake information to finish this faking process. A method often used is to create a new network packet for one or more information packets needed to deliver. This method will cause other system expenses and need the matching operations in the node server side. To avoid these drawbacks, we develop a new, efficient technique to deliver the information with an existing packet in Linux kernel. The salient feature of this mechanism is that it can prevent allocating new memory space and reduce system cost. The basic principle of this mechanism is that the sk_buff structure in Linux is not used entirely, thus allowing the unused memory space in the sk_buff structure to be used to deliver the handshake information. Two schemes are implemented to handle the case when the unused memory space is insufficient to contain the handshake information.

For generality, we design two modules: the pseudo-server module and the packet parser module. The pseudo-server provides the function of listening on multiple known ports at the same time. When a request (HTTP request, for example) comes, the pseudo-server will listen on the service port before the TCP connection has been established. The kernel portion of the pseudo-server will keep the selected three-way handshake information in a kernel hash table in the handshake process. When the pseudo-server notices that the TCP state is ESTABLISHED, which indicates that the TCP connection between the client and the network dispatcher has been established and the next packet will include the request content, the kernel portion will stop the listen process of this connection, and will clear the related data structures in the kernel. The packet parser module achieves generality by providing a common interface for developers to add other packet parser for different network services. In our prototype, we provide parser modules for HTTP and IIOP, so our Content Aware Scheduling System can support both HTTP and IIOP services.

PROTOTYPE IMPLEMENTATATION

There are three main parts in the proposed content aware scheduling system. In this section, we first present the design of the dispatcher module, and then discuss the Mix-LARD scheduling policy. Thirdly, another part of the CASS, the design of the node server module will be discussed. Last we will describe our implementation of configuration and controlling interface for CASS.

Network Stack of the Network Dispatcher

CASS is implemented at the IP layer. Network services must be configured in advance to inform CASS of the incoming packet for this service. When a new packet arrives, the packet will first be sent to the CASS for processing in IP layer, where the check module of CASS will check whether the coming service request has been configured. If not, this packet will be sent up to the normal TCP/IP stack where it will be processed directly. If this network service has been configured, this packet will be sent to the CASS for processing. Figure 2 depicts the network stack of CASS in the network dispatcher.

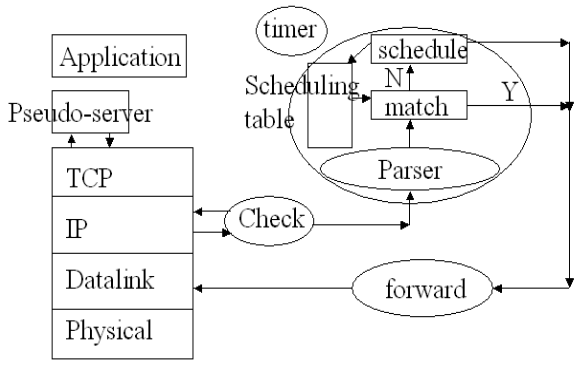

Figure 2 Network Stack of the Network Dispatcher

If the TCP state of this connection has not reached the ESTABLISHED state, the incoming packet will be sent up to the pseudo-server module for three-way handshake processing. If the TCP connection is ESTABLISHED, this packet will be delivered to the parser module. Based on service types, different parser will be called. The parsers' responsibility is to acquire the request's content. For example, if the service is WWW, the request's content is the URL address; if the service is based on IIOP, the request's content is the parameter of service function. After the parser module, the packet is delivered to the match module which looks up the scheduling table according to the request's content. If the same request's content is found in the scheduling table, implying that the same request has been scheduled before and it might still be in he node server's main memory, the request will be forwarded to the node server. The timer of CASS is used to delete old items

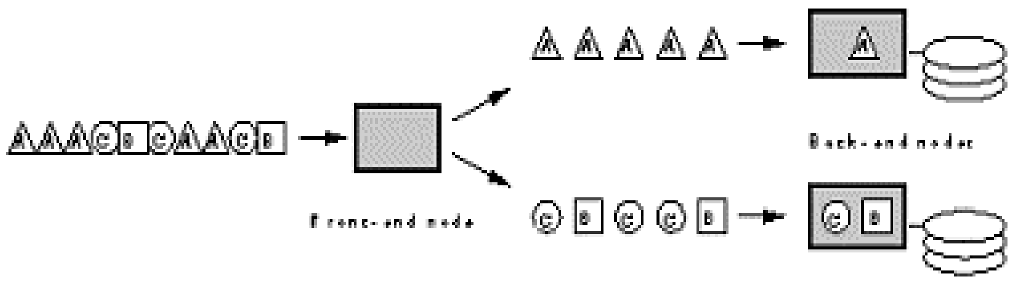

Figure 3 LARD Scheduling Policy

from the scheduling table. If a request is scheduled again, the timer of the request's "content hash table column" will be refreshed. If a request's content can't be found in the scheduling table, the schedule will use the schedule policy module to decide which node server to choose from. More details of the schedule policy will be discussed in second subsection.

After CASS has decided the target node server, the next step is to forward the client's request packet and handshake process information to the selected node server. The technique to deliver the handshake information along with the forwarded request packet, described in Section 3, is implemented in the data link layer. With this technique, the request packet and the necessary handshake information can be delivered together to the selected node server. The processing in the node server will be discussed in third subsection.

Mix-LARD Scheduling Policy

A new, mix scheduling policy called Mix-LARD will be discussed in this section. This scheduling policy is based on LARD. The basic principle of LARD is illustrated in Figure 3. In the figure, some web requests are arriving whose types are A, B or C, separately. According the allocation made *a prior*, each request will be forwarded to a selected node server for this service. For example, when a type A request arrives, this request will be forwarded directly to the node server designated to handle "A" service request, etc. Thus, the services with identical request content can be scheduled to the same node server, thereby enhancing the main memory cache hit rate at each node server by virtue of increased locality. Because the data fetching rate is much higher when the data is in the main memory than when the data is in hard disk, the cluster performance can be enhanced significantly.

Although LARD scheduling policy has many advantages, it has two main disadvantages:

(1) The allocation must be decided *a prior*, and the data must be allocated to different node servers. Hence, it is inflexible, and difficult for the user to configure or reconfigure.

(2) LARD policy can only support static web services, unable to support dynamic web requests and other protocols.

Therefore, we presented a hybrid scheduling policy, called the Mix-LARD. The first and most important feature of Mix-LARD is that it can support both dynamic and static web requests at the same time. To do so, a smart unit is embedded into the HTTP scheduling module. The smart unit is responsible to check the types of clients' requests. If a dynamic request is coming, the smart unit will find it and say to the scheduler: "Hey, it is dynamic. Use dynamic scheduling policy please." And if a common static HTTP request is coming, static scheduling policy will be called. This smart unit in CASS is implemented in Linux kernel, and its job is just to check some bits for every coming packet, so it's fast. It can work well without impairing the system's performance.

The second feature is that the scheduling allocation can be changed through the refreshing operation with the content aware Hash table's timer. To understand this feather, let's imagine the scheduling allocation table as a cache. By invaliding some stale data, we can enhance the hit rate of the scheduling allocation table.

Mix-LARD policy also can support many network protocols except HTTP, such as IIOP, etc. We implement this feather by providing the common programming interface for other possible network services.

Node Server Processing

In our content aware scheduling system, the responsibility of a node server is to receive the request packet that is forwarded by the network dispatcher, restore the forwarded request packet and fake the three-way handshake process to the user level network service programs like Apache. Figure 4 depicts the processing flow in a node server.

The packet receive module is implemented in the TCP/IP stack of a node server in the proposed CASS. This module's function is to receive the forwarded request. If it finds a packet that has CASS processing tag, this request will be passed to the packet resume module. The packet resume module will separate the three-way handshake information from the forwarded packet and restore the original client request packet. After that, the forwarded

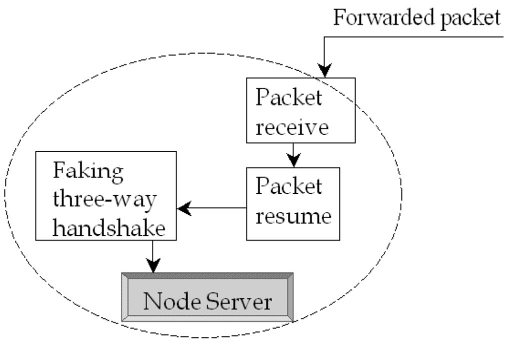

Figure 4 Processing Flow in Node Server

client request and three-way handshake information will be delivered to the module responsible for faking three-way handshake. In the faking module, some simulated operations will be done to stimulate the state changing of the TCP finite state machine. After these operations, the web service programs (such as Apache, etc.) will not notice that the client hasn't done three-way handshake process with it. It will handle the client's request and return the answer to client directly.

Configuration and Controlling Interface

Our CASS has supplied a configuration and controlling interface for administrators. To configure or control our CASS, an administrator can operate the web sever directly or just telnet into the Linux operating system in the web server to execute some commands. Some modules in CASS are responsible for this.

DISCUSSION OF CASS APPLICATIONS

CASS supports many TCP-based network services. In this section, we present two examples to show its application. The two examples are its support of IIOP (Internet Inter-ORB Protocol) and of cluster cache server.

CASS Support of IIOP

Because IIOP is the communication protocol that is used in CORBA program, we implement a simple CORBA program to show the support to IIOP. Our implementation includes a CORBA client, two CORBA node servers and a network dispatcher in which CASS is installed. A CORBA service called "echo" is running on each CORBA node server. This service's function is to print "server" in the terminal of the node server when it receive the client's request. According to the CORBA white paper, before the client sends request to the cluster, it must first get the IOR information that indicates necessary CORBA server information. When the CORBA client's request arrives at the network dispatcher, CASS will parse the request's content and use the Mix-LARD policy to select a CORBA node server. Then the request packet will be forwarded to the selected CORBA node server. After the node server receives the client request and the three-way handshake information, this CORBA node server will print "server" information in the terminal of this node server. It is because CORBA uses IIOP as its communication protocol that this test can show CASS's support to IIOP. Figure 5 illustrates the main idea of this test.

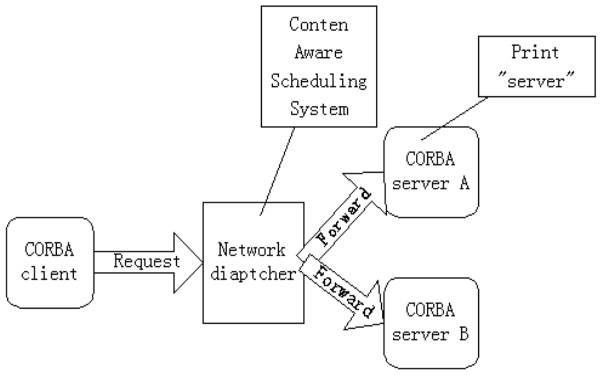

Figure 5 Support of IIOP

CASS Support of Cluster Cache Server

Cache server can be a PC plus the cache server software (such as Squid, Inktomi, etc), or it can be a special cache server like CacheFlow. In contrast to these cache server solutions, cluster cache server can use cluster architecture to provide support to cache services. Although the service type is different, cache server has the same architecture as cluster web server: the network dispatcher receives the client's request and forwards it to a selected node server. By adopting CASS, cluster cache server can gain the advantages of content aware scheduling policy: it can deploy different cache materials in different node servers. The cache server prototype implemented in this study uses the "CASS + Squid Cache server" scheme, and it works properly.

PERFORMANCE EVALUTATION

In this section, we present performance evaluation results obtained with our prototype CASS cluster. While the scalability test of the proposed CASS cluster is presented in the first subsection, the second subsection provides experimental results on the processing response time performance. Finally, the third subsection compares the performance of our CASS prototype with Linux Virtual Server—an example for IP level scheduling system.

Scalability of CASS

We use Webbench 4.1 to test the scalability of CASS cluster. The test environment includes four clients, one controller. The responsibility of clients is to send http requests to the cluster web server steadily, and the responsibility of controller is to send commands to every client to start the test and at the end of the test receives the clients test data and computes statistics on them. Webbench can test the throughput of the cluster, and our testing result is depicted in figure 6. In this test, the number of node servers increases from 1 to 6. In this scale, the throughput

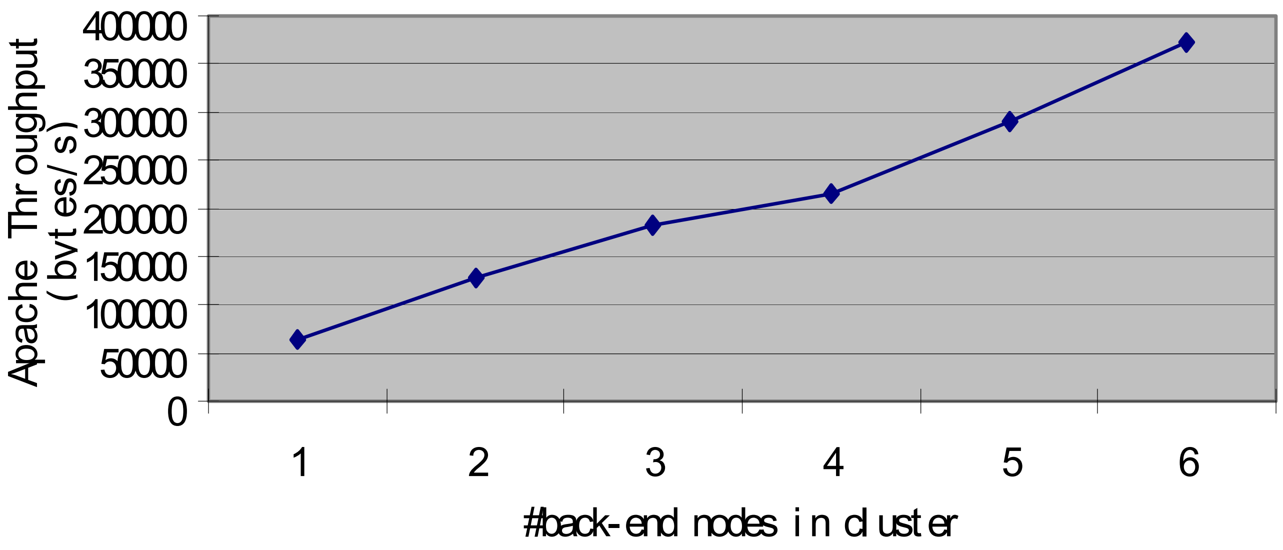

Figure 6 Scalability of CASS

of CASS cluster appears to scale almost linearly.

Processing Delay of CASS

This sub-section reports the test result on the processing delay of CASS. The testing tool used is httperf. The baseline delay was established by testing with one server with CASS. The processing delay with CASS minus as the baseline delay results in the CASS's delay value. In this experiment, the baseline server was found to process 135.4 connections per second, resulting in a per connection processing delay of 1000ms/135.4 = 7.39ms. Similarly, the CASS cluster with one server is measured to have a per connection processing delay of 1000ms/66.2 = 15.11ms/. Thus, the delay caused by CASS is 15.11 – 7.39 = 7.72ms. In general, the processing delay in a WAN environment is larger than 50ms, so the processing delay of CASS is acceptable. This is also true in LAN environment for end users.

Performance Comparison with Linux Virtual Server

Unlike CASS, Linux Virtual Server uses the IP level scheduling policy that select a node server without considering the request's content. As mentioned earlier, one of the advantages of CASS is that it can enhance the hit rate of node servers' main memory cache by taking advantage of locality. This performance advantage of CASS, however, comes at the expense of the higher system overhead of the network dispatcher compared with the Linux Virtual Server because CASS needs to do extra work related to request's content. In this experiment, we assess the performance impact of these two factors.Let α denote the communication delay of two PC, so the communication of the cluster's "client — network dispatcher — node server" architecture will be 2α. The processing delay of Linux Virtual Sever is assumed to be β, the processing delay of CASS to be β'. Suppose that data fetching delay from main memory is γ and the data fetching delay form hard disk is δ. Finally, the operating system processing delay is assumed to be ε.

When the test set is big enough to overflow the one-node server's main memory capacity, data must be fetched from the hard disk if Linux Virtual Server is adopted. On the other hand, if CASS is used as the dispatching module, data can be cached in different node server's main memory, thus avoiding the fetching delay from hard disk. In this case, the processing delay of Linux Virtual Server can be expressed in equation 1 below:

$$D_{lvs} = 2\alpha + \beta + \delta + \varepsilon \cdots\cdots\cdots\cdots\cdots (1)$$

The processing delay of CASS can be expressed in equation 2 below:

$$D_{cass} = 2\alpha + \beta' + \gamma + \varepsilon \cdots\cdots\cdots\cdots\cdots (2)$$

Equation 1 can be transformed to equation 3 as follows:

$$D_{lvs} = (2\alpha+\beta+\gamma+\varepsilon)+(\delta-\gamma)\cdots\cdots\cdots\cdots (3)$$

In equation 3, （2α＋β＋γ＋ε）is the processing delay of the Linux Virtual Server when it needs not fetch data from hard disk directly. So, the quotient of the processing delay of the Linux Virtual Server with that of CASS can be expressed as equation 4 below:

$$D = \frac{D_{lvs}}{D_{cass}} = \frac{(2\alpha+\beta+\gamma+\varepsilon)+(\delta-\gamma)}{2\alpha+\beta+\gamma+\varepsilon} \cdots\cdots\cdots\cdots (4)$$

According to the test result in the second subsection, D_{cavs} in equation 4 is 15.11ms. By using the same method, the test shows that the processing delay of Linux Virtual Server without direct hard disk fetching is 1000ms/115.6 =

8.65ms/conn. In other words, equation 4 can be transformed to equation 5 as follows:

$$D = \frac{8.65ms + (\delta - \gamma)}{15.11ms} \cdots\cdots\cdots\cdots\cdots (5)$$

In equation 5, δ is the fetching delay when data is fetched directly from hard disk. The fetching time from hard disk includes three parts: the time it spends to locate the target track (i.e., seek time), the time it spends to find the target sector (i.e., rotation time) and the time it spends to fetch the data (i.e., transfer time). Among these three parts, average seek time dominates, so we use the average seek time as its fetching delay from hard disk. The hard disk used in our testing environment is IBM Deskstar 60GXP. The information in IBM's product homepage shows that the average disk seek time is 8.5ms. Also, because the value of γ is in the level of nanosecond [6], we can ignore it here. In this case, equation 5 can be transformed to equation 6 as follows:

$$D = \frac{8.65ms + \delta}{15.11ms} = \frac{8.65ms + 8.5ms}{15.11ms} = 1.135 \cdots (6)$$

Thus, the average processing delay of CASS is 13.5% lower than that of Linux Virtual Server because CASS can prevent fetching data from hard disk directly.

CONCLUSION AND FUTURE WORK

We have presented a Linux content aware scheduling system for network services. CASS can increase the node servers' main memory cache hit rate and enhance the cluster's performance. It is possible for CASS to support other TCP-based network services.

Our design employs a PC that acts as the central point of contact for the server on the Internet, and distributes the incoming requests to a number of back-ends. CASS is implemented in Linux kernel except for the pseudo-server module, whose responsibility is to provide general listening function to different network services. The implementation at node server side is to receive the forwarded packet from network dispatcher and to fake three-way handshake process to network service programs.

In terms of scalability, the proposed design shows almost linear scalability with the number of node servers ranging from 1 to 6. The processing delay of CASS is low compared with the common network delay. We analyzed the CASS's processing delay in comparison with a commonly used IP level dispatching technology—Linux Virtual Server and the result shows that CASS's processing delay is 13.5% lower than that of the Linux Virtual Server.

In future study, we will optimize the performance of the pseudo-server module and investigate other issues that affect the performance of CASS.

REFERENCES

1. Cisco's LocalDirector, http://www.cisco.com/warp/public/cc/pd/cxsr/400/tech/lobal_wp.htm
2. C. Yoshikawa, B. Chun, P. Eastham, A. Vahdat, T. Anderson, and D. Culler, "Using Smart Clients to Build Scalable Services", *Proceedings of the USENIX 1997 Annual Technical Conference*, Anaheim, California, January 6-10, 1997
3. D. Andresen, T. Yang, O. H. Ibarra, "Toward a scalable distributed WWW server on workstation clusters", *Journal of Parallel and Distributed Computing*, Vol.42, No.1, 10 April 1997, pp.91-100
4. E. Anderson, D. Patterson, E. Brewer, "The magicrouter, an Application of Fast Packet Interposing", technical report, http://www.cs.berkeley.edu/~eanders/projects/magicrouter/
5. E. Walker, "pWeb – A Parallel Web Server Harness", http://www.ihpc.nus.edu.sg/STAFF/edward/pweb.html
6. K. Hwang, *Advanced Computer Architecture, Parallelism, Scalability, Programmability*, Prentice Hall, 1999.
7. M. Aron, D. Sanders, P. Druschel, etc, "Scalable Content-Aware Request Distribution in Cluster-Based Network Servers", *Proceedings of 2000 USENIX Annual Technical Conference*, 2000
8. O. P. Damani, P. E. Chung, Y. Huang, etc, "ONE-IP: techniques for hosting a service on a cluster of machines", *Computer Networks and ISDN Systems,* 29, pp.1019-1027
9. R. S. Engelschall, "Balancing your Web site. Practical approaches for distributing HTTP traffic", *WEB Techniques*, Vol.3, No.5, pp.45-6, 48-50, 52
10. T. T. Kwan, R. E. McGrath, D. A. Reed, "NCSA's World Wide Web server: design and performance", *Computer*, Vol.28, No.11, Nov. 1995, pp.68 -74
11. V. Cardellini, M. Colajanni, P. S. Yu, "DNS dispatching algorithms with state estimators for scalable Web-server clusters", *World Wide Web* (Netherlands), Vol.2, No.3, pp.101-103, 1999
12. V. S. Pai, "Locality-aware Request Distribution in Cluster-based Network Servers", *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*, San Jose, California, October 1998
13. W. Zhang, S. Jin, Q. Wu, "LinuxDirector: a connection director for scalable Internet services", *Journal of Computer Science and Technology*, 15(6), 560-571

14. X. Zhang, M. Barrientos, J. B. Chen, M. Seltzer, “HACC: An Architecture for Cluster-Based Web Servers”, *Proceedings of the 3rd USENIX Windows NT Symposium*, July 1999