A Context-Based Request Dispatching Policy for Layer-7 Web Cluster

Chien-Hung Chen, Chun-Liang Hou, and Shie-Jue Lee
Department of Electrical Engineering
National Sun Yat-Sen University
Kaohsiung 80424, Taiwan, ROC
Email: jianhong@water.ee.nsysu.edu.tw

Abstract

Locality-aware request distribution (LARD) [2] is a dispatching policy commonly used in the front-end node of the Web cluster. The principle of LARD is to dispatch all requests for identical context to the same back-end server. Such principle increases the likelihood to find the requested context in the cache of the back-end server. However, LARD makes decision based on the number of TCP connections between the front-end node and each back-end server. Due to each connection does not always produce requests from clients, the decision made by LARD would not be totally accurate. In this paper, We propose an efficient context-based request dispatching policy to solve the problem of LARD. With our policy, back-end servers would send some state information such as input queue length and total service time of queue, back to the front-end node periodically. Obviously, the load of back-end server calculated based on feedbacks is more accurate than that only based on the numbers of TCP connections. In addition, we also develop two algorithms for packet classifier and admission control mechanism, respeclity. These two mechanisms are used to mantain performance for Web cluster and avoid Web cluster overloaded. The simulation result demonstrates that our policy can achieve high cache hit rates and good load balance on the back-end servers, and uses smaller number of table entries in the front-end node than LARD.

Keywords: Layer-4/7 switch; Web cluster; dispatching policy; packet classifier; admission control

1 Introduction

Performance, scalability and availability are critical for Web systems even receiving large numbers of request from clients. New generation of Web systems provides more complex services, and results in rapidly increasing traffic. To avoid raising response time of request and reducing throughput of Web system, an approach named distributed Web systems [1] [4] [11] is becoming a common choice for most popular Web sites. In distributed Web systems, with the replication of information across independent or coordinated mirrored servers, the request of clients assigned to a target Web server is typically carried out during the address resolution of the Web site name by the DNS mechanism or assigned manually by users. In addition, an architecture called cluster-based Web system (Web cluster) [10] that a pool of servers are tied together to act as a single unit has been proposed. Generally speaking, a Web cluster consists of one front-end node and multiple back-end servers connecting with each other by high speed LAN. Although a Web cluster consists of multiple nodes, it is provided with one site name and single virtual IP (VIP) corresponding to the address of the front-end node. For the Web cluster, only one single host address appears to the outside world. So that all incoming requests from any client would be received by the front-end node, and then the front-end node dispatch the received requests to the suitable back-end server.

In this paper, we consider that the Web cluster is more efficient than distributed Web system. With Web cluster, we can manage all back-end servers centralized, it is convenient to Web administrator to add or delete a back-end server of Web cluster. In addition, the front-end node can assign request based on the load of each back-end server in Web cluster. This would make the decisions more appropriate than distributed Web systems which the target server of request is chosen manually by users. Therefore, distributed Web system do not achieve good load balance among all back-end servers. In a word, Web cluster is more scalable and fault-tolerant than distributed Web system. Some Web cluster technologies have been proposed in [10] [11] and described in detail.

With Web cluster, the front-end node may use various dispatching policies to assign the incoming request to the back-end servers. We can classify dispatching policies into two kinds, static and dynamic [13]. The main difference between these two kinds is that static dispatching policies fail not consider any state information of the system. Random, round robin and hash function are commonly seen static dispatching policies. The advantage of static policies is simple to be implement and quick to make decisions for hundreds or thousands of requests per second to prevent the front-end node becoming the bottleneck of the Web cluster. However, static dispatching policies potentially make inefficient dispatching decisions.

Dynamic dispatching policies can make a good decision by analyzing the feedback information from servers or clients. In intuition, dynamic dispatching policies have the potential to outperform static dispatching policies. LARD is one of the famous dynamic dispatching policies and it analyzes the number of connections between the front-end node and each back-end server to index the load of back-end server. The front-end node would also inspect all incoming requests to aware which context is being requested. Then, LARD decides a suitable back-end server based on these information. Global memory system (GMS) [2] is another interesting policy belonging to dynamic policies. GMS improves Web cluster performance by adopting a global memory system on back-end servers so as to increase the entire hit rate of main memory cache of Web cluster. Although dynamic policies make more accurate decisions than static policies, we should still be careful to prevent the front-end node to become the bottleneck of Web cluster. Since the computational complexity of dynamic policies is higher than static policies and dispatching decisions are required to be made in real-time. But the solution of this drawback is out of the scope of our paper, and we will focus on the dispatching policy mostly.

The context-based dispatching policy we propose is also one of dynamic dispatching policies. The main difference between LARD and our context-based dispatching policy is that our dispatching policy does not maintain a large size of bind-table to record one-to-one mapping of contexts to back-end servers. More, our context-based dispatching policy uses server state informations, such as queue length and total service time of queue in back-end servers to decide a suitable back-end server for the incoming request. As we can see later, our policy outperforms LARD in terms of throughput and cache hit rate and achieves good load balance between back-end servers.

The remainder of the paper is organized as follows. In section 2, we outline the Web cluster technologies and focus on the front-end node, we also describe the operation of packet classifier and admission control in detail. In section 3, we will describe our context-based request dispatching policy in detail. In section 4, we introduce the simulation model and discuss the experimental results. The simulation result shows our policy can achieve better capacity than LARD, but use least memory in the front-end than LARD. In section 5, finally, we summarize the conclusion of this paper.

2 Web cluster architectures

Web cluster consists of a front-end node, responsible for request dispatching, and a number of back-end servers, responsible for request processing. The technology of Web cluster has been discussed

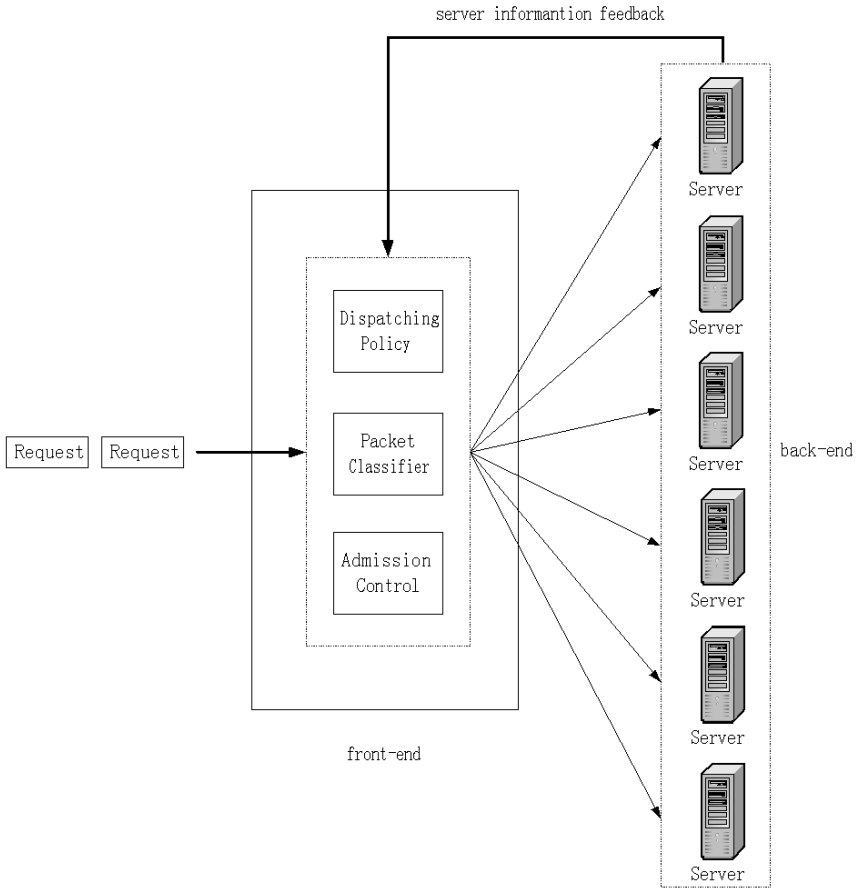

Figure 1: The architecture of Web cluster.

detailed in [10]. Such as layer four switching with layer two packet forwarding (L4/2), layer four switching with layer three packet forwarding (L4/3), and layer seven (L7) switching with either later two packet forwarding (L7/2) or layer three packet forwarding (L7/3). In this paper, we adopt layer-7 switch as the front-end node and L7/2 technology [10] has been used. Figure 1 shows the L7/2 Web cluster architecture. The front-end node of Web cluster is called dispatcher [10] or Web switch [13], retains transparency of the Web cluster for the users and dispatches all incoming request to the beck-end servers. From client's point of view, any request to the Web cluster is presented to the front-end node which acts as a agent for the Web cluster.

The front-end node includes three mechanisms which are dispatching policy, packet classifier and admission control, respectively. Dispatching policy is the focus of this paper, and we will describe ours in detail in section 3.

Once a request comes to the front-end node, the packet classifier has to determine whether request to establish a new TCP connection or not for this request. If the answer is true, the packet classifier checks two fields of packet header and the command of cookie stored in the client end against the existing member-table. It distinguishes an incoming request into different levels of priority and marks a priority value p to this request. The member-table provides a static priority value for particular clients. For instance, the member of this Web site would get a higher priority.

If the answer is false, the packet classifier looks up a match filter in the filter-table and assigns a priority value p to this request. The filter-table records state of existing TCP connections and dynamic adjust priority for each filter based on load of Web cluster and state of each TCP connection. Figure 2 shows the algorithm of our packet classifier.

```
While (true)
    fetch next request r and check the SYN field of TCP header;
    if (SYN == 0)
        look up the member-table ;
        assign priority p to this request ;
    else
        look up the filter-table;
        assign priority p to this request;
```

Figure 2: The algorithm of our paket classifier.

```
While (true)
    fetch next request r;
    if the server is not overloaded;
        ACK;
        if (SYN == 0)
            use three-way handshake to establish a TCP connection;
            insert a new filter for this connection in filter-table;
    else;
        check the priority p assigned by request classifier;
        if (p is higher than the lowest priority of the filter-table)
            ACK;
            if (SYN == 0)
                use three-way handshake to establish a new TCP connection;
                delete the lowest priority filter f from the filter-table;
                close the TCP connection of filter f;
                insert a filter for incoming request;
            else;
                reject;
    adjust the priority of each filter corresponding to state of each connection and system;
```

Figure 3: The algorithm of our addmission control.

For admission control, it takes the load of Web cluster and the priority that marked by pakcet classifer into account to decide whether to reject the incoming request or not. Once the Web cluster is overloaded, and the priority assigned by packet classifier to the incoming request which requests to establish a new TCP connection is higher than the lowest priority of filter in the filter-table, the filter in the filter-table that has lowest priority would be replaced by the incoming request, and the existing TCP connection for replaced filter would be closed. Then establish a new TCP connection for incoming request. If the incoming request belongs to a existing TCP connection, admission control decides whetehr to reject this request or not based on the priority p of this request.

If the Web cluster does not overload, all incoming requests would not be reject. The admission control maintains the filter-table and adjusts the priority of each filter based on the load of Web cluster and the state of each filter. The algorithm of admission control is shown in Figure 3.

We develope a packet classifier and an admission control algorithm to guarantee QoS of each TCP connection and avoid back-end servers to overload. We give an example to show the operation of these two mechanisms. Assume the incoming request r has source IP = 140.117.164.90, source

Src. IP	Src. Port	command of cookie	Priority
140.117. ∗ .∗	4147	command A	2
140.122.65.∗	*	command B	7
163.124.10.103	*	command C	3
210. ∗ . ∗ .∗	6521	command C	0

Table 1: The original member-table of the example.

Src. IP	Src. Port	recent issue time	delay time	Priority
140.117.164.108	4557	9:10	4.0	0
140.122.65.25	3987	3:12	5.6	5
210.149.106.141	1821	1:03	6.8	7
210.135.11.1	6521	0:05	4.1	7

Table 2: The original filter-table of the example.

port = 4147 and SYN = 0. The original member-table and filter-table are shown in Table 1 and Table 2, respectively. First, the packet classifier would look up the member-table and marks the request r a priority value $p = 2$. Then this request would be delivered to the admission control. If the Web cluster is not overloaded, admission control would insert one filter for this requet in the filter-table and establish a TCP connection for this request. The result of filter-table is showen in Table 3. Otherwise, it would replace one filter of the filter-table that has lower priority than the priority of incoming reuqest and close this TCP connection. Then establish a new TCP connection for incoming request. The result of filter-table is shown in Table 4.

The packet classifier and admission control is used to maintain the performance of Web cluster, avoid back-end servers overload. In addition, we assume all back-end servers have equally capability to handle a given request, and each back-end server could feedback some information to front-end periodically, such as CPU utilization, input queue length, and the number of active connections, front-end node would decide suitable back-end server based on this information.

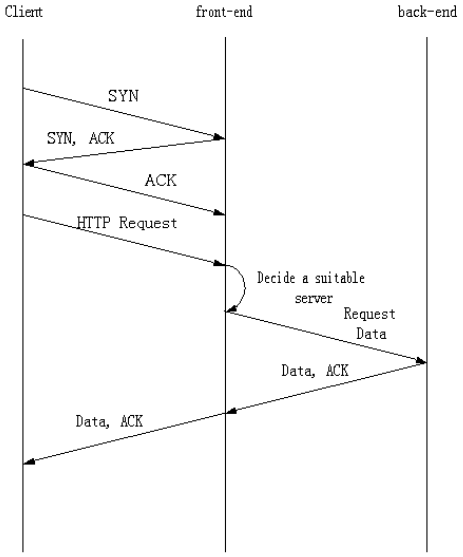

Figure 4: Operation of layer-7 switch.

Src. IP	Src. Port	recent issue time	delay time	Priority
140.117.164.90	4147	initiate	0.0	2
140.117.164.108	4557	9:10	4.0	0
140.122.65.25	3987	3:12	5.6	5
210.149.106.141	1821	1:03	6.8	7
210.135.11.1	6521	0:05	4.1	7

Table 3: The filter-table of the example after insert a new filter.

Src. IP	Src. Port	recent issue time	delay time	Priority
140.117.164.90	4147	initiate	0.0	2
140.122.65.25	3987	3:12	5.6	5
210.149.106.141	1821	1:03	6.8	7
210.135.11.1	6521	0:05	4.1	7

Table 4: The filter-table of the example after replace a filter.

Layer-7 switch operates at application layer of OSI (Open System Interconnection) model, that is, it would inspect the URL field and other HTTP request header information of the incoming request. The operation of L7/2 architecture is shown in Figure 4. Whenever a client tries to issue a request to the Web cluster, a TCP connection between the front-end node and client should be established first. This results in an exchange of SYN and ACK packets to complete the TCP's three-way handshake procedure [15]. Here, we assume that HTTP version 1.1 has been used in our architecture. HTTP version 1.1 [17] allows requests pipeline, and multiple HTTP request can use the same TCP connection. This differs with HTTP version 1.0 [16] on that HTTP request and TCP connection are correspondence relation in HTTP version 1.0, which means HTTP request and TCP connection are one-to-one mapping.

After a connection established between client and front-end node, front-end node can examine the subsequent HTTP request, consider the state information of back-end servers to perform context-based dispatching, and then dispatch the request to the suitable back-end server based on MAC address. When a back-end server receives a request from the front-end node, it will handle the given request and send back the result to front-end node.

3 Context-based request dispatching policy

The major challenge in designing a Web cluster with request dispatching policy is to achieve good load balance and high cache hit rate on back-end servers. For load balance, we hope that all of the back-end servers have equal load and no back-end server becomes idle under all kinds of situations. For cache hit rate, we attempt to dispatch requests that have the identical requesting context to the same back-end server. Without doubt, such attempts will increase the hit rate of cache on each back-end server. One simple and famous approach to dispatch requests is round-robin (RR) dispatching policy. With RR dispatching policy, the front-end node could not aware what context is being requested by the incoming request. It can reduce the dispatching complexity of the front-end node, and also prevent the front-end node to become the bottleneck of Web cluster. Weight round robin (WRR) dispatching policy is evolved from RR, the incoming requests are dispatched in round robin manner, weighted by the load of each back-end server. The load is measured based on some state information of the back-end servers, for instance, CPU utilization, memory utilization, or the number of connections. With RR and WRR dispatching policies, each back-end server receives

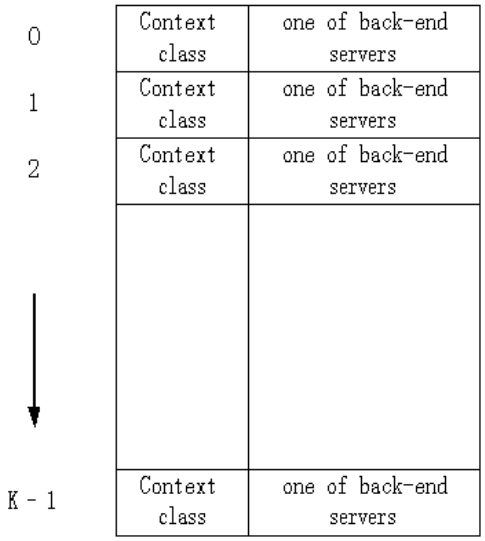

Figure 5: Structure of the bind-table.

similar numbers of requests. Since each server has to fit the entire working set, they could never hit the high cache hit rate.

For context-based request dispatching policy, not only the state information of back-end servers, the front-end node should also aware the context requested by the incoming request. Based on this information, the front-end node would choose a suitable back-end server and dispatch the incoming request to it. LARD is a context-based request dispatching policy proposed in [2], which dispatches incoming requests in a manner that can achieve high hit rate of cache as well as good load balance. With LARD, the front-end node must maintain a one-to-one mapping table to record each target and the corresponding back-end server. The term target means a specific object requested from servers, for instance, a target is specified by a URL and HTTP GET command for an HTTP server. LARD makes dynamic decisions on the basis of the load of the back-end servers. The load of each back-end servers in LARD is measured as the number of connections associated between the front-end node and each back-end server. But it is not surely accurate, since each connection does not always produce requests from clients. To alleviate the problem, we present a more efficient policy that can achieve exceeding capability of LARD and less memory use in the front-end node. The context-based request dispatching policy takes the state information of back-end servers, such as request service time and input queue length into account. It makes decision accurate.

In order to increase the hit rate of cache in back-end servers, the intuition is to dispatch incoming requests having the same requested context to identical back-end server. Consequently, we should record the back-end server which the prior context requested dispatch to. With our policy, first, we classify entire contexts into K classes. The front-end node should have a bind-table with K entries that records one-to-one mapping of K classes to one of the back-end servers. The structure of bind-table is shown in Figure 5.

Initially, each field of bind-table is $NULL$. Then the front-end node should decide the suitable server that mapping of each context class period, according the state information of back-end server. To clarify the description of our policy, we first define some terms.

- N: the number of back-end servers
- n: the candidate back-end server of incoming request
- R_i: the number of requests that belongs to the same context class R in the serve i.

parameters	value
K	4
N	3
A_1	1
A_2	2
A_3	1
B_1	1
B_2	0
B_3	0
C_1	0
C_2	1
C_3	1
D_1	0
D_2	0
D_3	0
q_1	2
q_2	3
q_3	2
T_1	15
T_2	22
T_3	17

Table 5: Parameters value of the example.

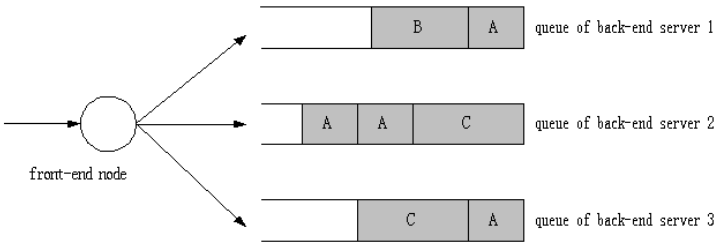

Figure 6: A Web cluster example.

- q_i: the input queue length of server i.
- T_i: the total service time of the input queue in back-end server i.

When the front-end node receives the state information that be sent from back-end servers, it should calculate a candidate back-end server in each context class and record the result in the bind-table. The candidate server in each context class is decided as follows. In the first place, the normalized request rate of server i in class R is calculated by Eq 1.

$$H_{R,i} = \frac{\frac{R_i}{q_i}}{\Sigma_{n=1}^{N} \frac{R_n}{q_n}} \tag{1}$$

$$n_R = \begin{cases} NULL & \text{if } \forall i, H_{R,i} = 0 \\ \arg\max_i(H_{R,i}) & \text{if } \exists H_{R,i} \neq 0 \end{cases} \tag{2}$$

Where $i \in \{1, 2, ..., N\}$ and $R \in \{1, 2, ..., K\}$. Next, by Eq 2, for each context class, the candidate server will be the server with the maximal value of $H_{R,i}$. If all $H_{R,i} = 0$, there is no existing suitable server for this context class, the n_R would be assigned $NULL$.

We give an example to show how to decide a suitable back-end server n of each contexts class. Figure 6 shows the situation for the Web cluster and Table 5 shows the value of parameters defined

context class	target server
A	2
B	1
C	2
D	NULL

Table 6: The bind-table of the example.

While (true)
  fetch next request r that belongs to class R and lookup bind-table;
  if (the field of bind-table == NULL)
    n = {*server that has the least load of back-end servers* };
  else
    n = {*server recorded in bind-table* };
    if (load of server n - least load of back-end server > T_{load}*)*
      n = {*server that has least load of back-end server* };
  send r to n;

Figure 7: The algorithm of our dispatching policy.

above. We assume service time of context A, B, C, D is 5, 10, 12, 15 sec, respectively. The bind-table of this situation is shown in Table 6.

When the front-end node receives a request from a client, it would inspect what context class being requested. Then look up the bind-table to find the mapping back-end server. If the field of a context class in bind-table is $NULL$, there is no requests belonging to this context class in all input queue of back-end servers. In this situation, the front-end node will choose a back-end server that has the least load to dispatch incoming requests. The load is measured as the total service time T_i of input queue in the back-end server.

Figure 7 presents the algorithm of our context-based dispatching policy. We define the T_{load} as the threshold of load.

The front-end node find the candidate back-end server n by searching the bind-table.

It must examine whether subtract least load of back-end server from load of n is greater than T_{load}. If the answer is true, the front-end node will dispatch incoming request to back-end server that has the least load. Otherwise, the incoming request will be dispatched to the server n. T_{load} should be chosen low enough to avoid load imbalance between back-end servers, which may lead to a high packet loss. Adversely, T_{load} should be chosen high enough to increases the hit rate of cache. Setting for T_{load} involves a tradeoff because it should be low enough to tolerate load imbalance and high enough to increase hit rate of cache. In our simulation, we have found settings $T_{load} = 600$ sec to give good performance across all situation we tested.

4 Simulation and discussion

In this section, we conduct our simulations using the OPNET Modeler. We define three levels of service which are high, medium and low level, respectively. Multimedia and voice belong to high level and this level of service needs QoS guarantee to prevent delay jitter, and may need more processing time in back-end servers. CGI, ASP and Java scripts belong to medium level. This type of context needs more computing time in CPU and may access database. Image and text contexts belong to low level, which needs least processing time. In our simulation, their percentage of entire

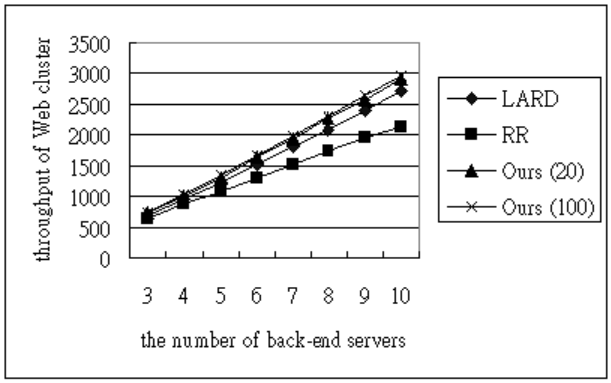

Figure 8: Throughput of Web cluster which has 1000 contexts.

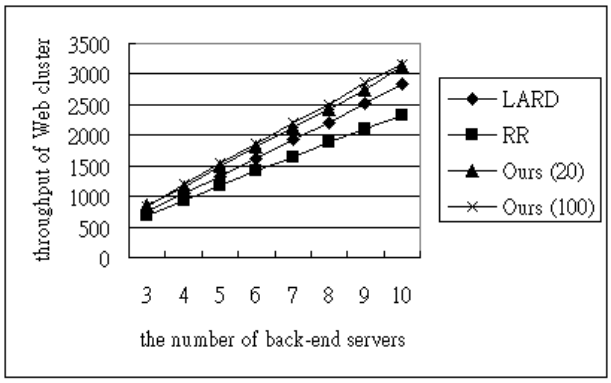

Figure 9: Throughput of Web cluster which has 500 contexts.

contexts are set to 60%, 25%, and 15%, respectively.

We assume there exist 6000 clients producing requests to Web system at random time, and the interval of mean time between any two requests depends on poisson distribution. For the mechanism of admission control, we set $T_{high} = 65$, and $T_{low} = 55$. That is, the front-end node limits the total number of connections to the value $S = (N-1) * T_{high} + T_{low} - 1$. As the replacement strategy of cache on the back-end servers, LRU (least recent used) strategy has been used in this simulation. In order to simplify our simulation, we assume the size of input queue in back-end servers is infinite. It means that we would never consider the requests dropped by a back-end server. And we fix the priority of admission control, do not consider the state of filter-table impermanent.

The throughput of Web cluster is shown in Figure 8 and Figure 9, with the entire context of Web cluster = 1000 and 500, individually. It can be seen that the throughput of RR is lower than our policy and LARD. Obviously, the reason is that RR does not achieve high hit rate of cache. The hit rate of cache is shown in Figure 10 and Figure 11. As we can see, with RR, the hit rate of cache does not become higher even if the number of back-end servers is increasing. As expected, both the throughput and hit rate of our policy exceed LARD. The result shows if we classify entire contexts at least into 20 classes, the throughput and hit rate will outperform LARD. Moreover, we only need 20 entries of bind-table in the front-end node, and is greatly less than LARD. The hit rate of cache will be higher and higher with the number of context classes we classified. This results in the increasing throughput of Web cluster.

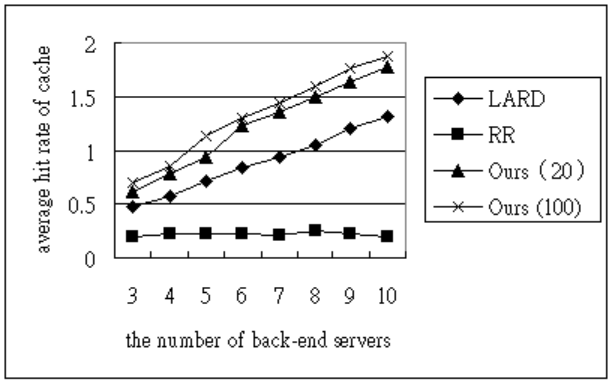

Figure 10: Average hit rate of cache that Web cluster has 1000 contexts.

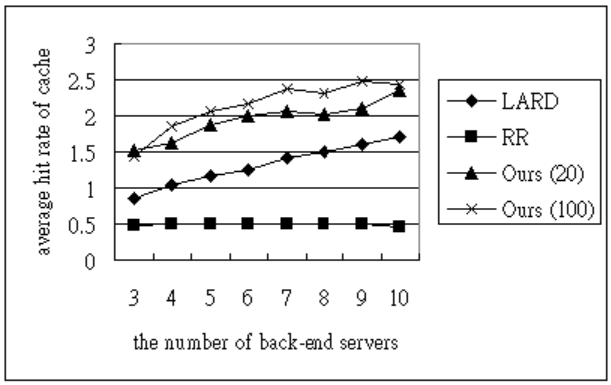

Figure 11: Average hit rate of cache that Web cluster has 500 contexts.

5 Conclusion

Web cluster architectures are becoming very popular for supporting Web sites with large numbers of servers. In this paper, we propose an efficient context-based dispatching policy that achieves high cache hit rate and good load balance. The simulation result shows the performance of our policy exceeds LARD. Specifically, we use less memory in front-end than LARD. In our on-going work, we consider the input queue size being finite. Therefore, the context-base dispatching policy should consider the drop of requests in back-end server. In addition, the service QoS of three levels could be guaranteed. The main challenge is to develop a strategy to reduce the drop rate in the back-end server.

References

[1] Michele Colajanni, Philip S. Yu, and Daniel M. Dias, "Analysis of Task Assignment Policies in Scalable Distributed Web-Server Systems," *IEEE Trans. on Parallel and Distributed Systems,* vol. 9, no. 6, pp. 586-600, June 1998.

[2] Vivek S. Pai, Mohit Aron, Gaurav Banga, Michael Svendsen, Peter Druschel, Willy Zwaenepoel, and Erich Nahum, "Locality-Aware Request Distribution in Cluster-based Network Servers," *Proc. ACM Arch. Support for Progr. Languages Conf.,* pp. 535-544, October 1998.

[3] Emiliano Casalicchio and Michele Colajanni "A client-aware dispatching algorithm for web clusters providing multiple services," *Proc. ACM World Wide Web Conf.,* pp. 535-544, May 2001.

[4] Valeria Cardellini, Emiliano Casalicchio, Michele Colajanni, and Philip S. Yu "The state of the art in locally distributed Web-server systems," *ACM Computing Surveys,* vol. 34, no. 2, pp. 263-311, June 2002.

[5] Joel L. Wolf and Philip S. Yu "On balancing the load in a clustered web farm," *ACM Trans. on Internet Technology,* vol. 1, no. 2, pp. 231-261, November 2001.

[6] Narasimha Reddy, A.L. "Effectiveness of caching policies for a Web server," *Proc. IEEE High-Performance Computing Conf.,* pp. 94-99, 1997.

[7] Abdelzaher T.F. and Bhatti N. "Web server QoS management by adaptive content delivery," *IEEE Quality of Service Workshop,* pp. 216-225, 1999.

[8] Li-Chuan Chen and Hyeong-Ah Choi "Approximation algorithms for data distribution with load balancing of web servers," *Proc. IEEE Cluster Computing Conf.,* pp. 274-281, 2001.

[9] Chu-Sing Yang and Mon-Yen Luo "A content placement and management system for distributed Web-server systems," *Proc. IEEE Distributed Computing Systems Conf.,* pp. 691-698, 2000.

[10] Schroeder T., Goddard S., and Ramamurthy, B. "Scalable Web server clustering technologies," *IEEE Network,* vol. 14 no. 3, pp. 38-45, May/June 2000.

[11] Cardellini V., Colajanni M., and Yu, P.S. "Dynamic load balancing on Web-server systems," *IEEE Internet Computing,* vol. 3 no. 3, pp. 28-39, May/June 1999.

[12] Iyengar A., Challenger J., Dias D., and Dantzig, P. "High performance Web site design techniques," *IEEE Internet Computing,* vol. 4 no. 2, pp. 17-26, March/April 2000.

[13] Valeria Cardellini, Emiliano Casalicchio, Michele Colajanni, and Salvatore Tucci "Mechanisms for quality of service in Web clusters," *Computer Network,* vol. 37 pp. 761-771, 2001.

[14] Ying-Dar Lin, Huan-Yun, and Kuo-Jui Wu "Ordered lookup with bypass matching for scalable per-flow classification in layer 4 routers," *Computer Communications,* vol. 24 pp. 667-676, 2001.

[15] W. Richard Stevens "TCP/IP Illustrated Volume 1," *Addision-Wesley,* 1994.

[16] T. Berners-Lee, R.Fielding, and H. Frystyk "Hypertext Transfer Protocol - HTTP/1.0," *http://www.w3.org/hypetext/WWW/Protocos/.*

[17] T. Berners-Lee, R.Fielding, H. Frystyk, J. Gettys and J.C. Mogul "Hypertext Transfer Protocol - HTTP/1.1," *http://www.w3.org/hypetext/WWW/Protocos/.*