# Scalable and Decentralized Content-Aware Dispatching in Web Clusters

Zhiyong Xu
Suffolk University
zxu@mcs.suffolk.edu

Jizhong Han
Chinese Academy of Sciences
hjz@ict.ac.cn

Laxmi Bhuyan
University of California, Riverside
bhuyan@cs.ucr.edu

***Abstract*—**
*In this paper, we propose a novel and efficient content-aware dispatching algorithm. Our approach eliminates the potential bottleneck and the single point of failure problems completely by using totally decentralized P2P architecture. It is scalable, the system throughput increases nearly linearly with the increased number of servers. Meanwhile, it does not introduce heavy communication overhead among back-end servers which appeared in the previous decentralized mechanisms. Our simulation results show that our approach is superior to the previous solutions.*

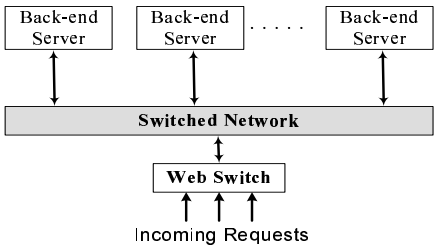


Fig. 1. Typical Web Cluster Architecture

## I. Introduction

Internet experienced fascinating growth in the past decade. World Wide Web (WWW) becomes one of the major information sources. Cluster based web server system (*web cluster*) becomes one of the most prevailing mechanisms to satisfy this need. It has been proved to be cost-efficient web service architecture which can provide high performance at low investment costs. In general, a web cluster is consisted of a number of commodity computers which hosted in a single location. It uses a front-end web switch as the access point for the incoming client requests. As shown in Figure 1, the web switch accepts the incoming requests and selects back-end servers to fulfill these requests according to the dispatching algorithm. Typically, this algorithm should secure the loads on the back-end servers is balanced and the average service time per request is minimized. Thus, the policy adapted in the dispatching algorithm has great influence on the system overall performance. Depends on which type of web switch is used in the web cluster, request dispatching algorithms can be categorized into two models: Content-blind approach and Content-aware approach.

In content-blind approach, a layer-4 web switch is deployed. The layer-4 switch does not check the content of the requests. It can only use simple dispatching algorithms such as Round Robin (RR), Weighted Round Robin (WRR) etc. It can be viewed as the sole load-based algorithms. These algorithms are simple, efficient and fast. However, because of the lack of the content information, content-blind algorithms can only dispatch requests based on the workloads on back-end servers. Normally, the server which has the lowest workload is chosen. The web switch is not aware of the contents in the back-end servers' main memory buffers. A request which might be able to be satisfied with the buffer cache on one server is very likely sent to another server instead.

In contrast, in the content-aware approach, a layer-7 web switch always inspect the layer-7 information for each request and take this information into consideration when choosing which back-end server should handle a given request. The system can take client request information as well as server workload status into consideration and try to achieve the balance between the load sharing and the cache affinity property. Locality-Aware request distribution (LARD) [1] and HACC [2] are the most popular content-aware approaches.

Compare to the content-blind approach, the content-aware approach can afford significant performance improvement. However, such a promising approach comes at a cost, it has a serious drawback associated with its benefits: the processing of the layer-7 information generates much more operational overhead on the front-end switch.

Here, each request must pass through the whole TCP/IP protocol stack. Thus, content-aware approach is not scalable. A layer-7 web switch can not support as many back-end servers as a layer-4 web switch which only need to process the layer-4 information. Thus, in the web clusters which contain a small number of back-end servers and the request traffic intensity do not go beyond the maximum processing power of the front-end layer-7 switch, the content-aware approach is superior. While in case there are a large number of back-end servers or system have been overwhelmed with a high volume of requests, a layer-4 web switch with content-blind dispatching algorithms may be more efficient. Another important issue for both content-blind and content-aware algorithms is: the usage of the single layer-4/Layer-7 web switch causes a single point of failure problem. The web switch failure will lead to the whole system out of service.

To address these problems, in this paper, we investigate scalable software architecture for content-aware approach. We propose a new request dispatching algorithm for web clusters. Similar to the current content-aware mechanisms, our approach aims to achieve the accurate dispatching decision by tracking the layer-7 information. Our approach is intended to improve the system scalability by distributing the dispatching workloads across all the servers. The single incoming web switch is eliminated. Every server is used as the combination of a web switch, a request dispatcher and the requested file provider. The system workload is evenly distributed on all the servers. For N servers, a server only takes 1/N portion of all system workloads. It is simple, efficient and totally decentralized. The main contributions of our algorithm are:

1) It eliminates the potential bottleneck and the single point of failure problem on the web switch by using totally decentralized service architecture. Every server can receive the incoming request;
2) It solves the heavy maintenance overhead problem in the previous decentralized solutions such as L2S. We use a distributed hash table (DHT) based dispatching algorithm, no inter-server communication overhead is introduced;
3) It is a highly scalable mechanism in which the system throughput increases nearly linearly with the addition of more servers.

The rest of the paper is organized as follows. In Section II, we give the brief description of the content-aware dispatching algorithms and the existing problems. In Section III, we present the detailed system design of our algorithm. In Section IV and V, we introduce the experimental environment used in our simulations, and compare the performance of our approach, with other solutions. In Section VI, we describe some related works. Finally, in Section VII, we conclude the paper.

## II. Problems in Content-Aware Switch

### A. Scalability Problem

The current web cluster architecture has the scalability problem. As we mentioned previously, in content-aware dispatching algorithms, the layer-7 front-end web switch must inspect the entire HTTP request and then make the distribution decision based on this client side information. Since the processing for each request must go through the whole TCP/IP protocol stack, it introduces heavy processing overhead and limits the system throughput. To reduce the overhead on the web switch, TCP splicing [3] [4] and TCP handoff [5] [6] techniques have been proposed. In TCP splicing, the response of the request does not go through the whole TCP/IP stack. While in TCP handoff, the response messages are sent back to the clients directly and avoid the web switch completely. However, even with these techniques, system throughput is still limited by the single front-end because it has to deal with every incoming request. In [6], M. Aron et. al. conducted experiments to prove this.

### B. Scalable Content-aware Approach

In [6], Aron et. al. presented a scalable content-aware approach to address the above problem. The functionality of the request dispatching is divided into *distributor* and *dispatcher*. The dispatcher is the component that implements the request distribution strategy. The distributor interfaces with the clients and implements the mechanism that distributes the client requests to back-end servers. The distributor is co-located with the back-end servers, only the dispatcher is a centralized facility. The experiments result showed this approach can greatly improve the system scalability.

The above solution reduces the overhead on the front-end switch, and system can achieve better scalability. However, the single dispatcher can still be the potential bottleneck with a large number of back-end servers. For example, in a web server systems with tens or hundreds of thousands back-end servers, a single dispatcher is not able to support such a huge number of servers. Another serious problem of such centralized architecture is the reliability. In case the central dispatcher is dead, the whole system is out of service.

### C. Locality and Load Balancing (L2S) Approach

The best way to address both the scalability and reliability problems in web clusters is to deploy a totally decentralized architecture. L2S [7] used such architecture. In L2S, there's no front-end or central facility any more. Each back-end server in the cluster can accept the request, and make the decision to deal with the request by its own or forward to another server. The experimental results shown L2S has very good scalability. The whole system

throughput increases nearly linearly as more servers are added in the system. It also has better reliability than pervious solutions. A back-end server failure does not affect the web cluster very much since some other servers can take its responsibility. The system performance just degrades gracefully.

However, this approach also has its drawback. In L2S, each server must communicate with other servers periodically and notify the peer servers the current contents in the main memory buffer cache. Based on this information, each server can make the optimal dispatching decision. This introduces high communication overhead and limits its efficiency with a large number of servers in the cluster. Thus, just like LARD, it can not support too many back-end servers in reality. More sophisticated algorithms are needed to solve this problem.

## III. System Design

We propose a new content-aware dispatching algorithm to address the drawbacks in the previous solutions.

### A. System Architecture

In our approach, the request dispatching is done in two steps. As shown in Figure 2, in the first step, each server is accepting client requests from an outside router or switch (a network devices, not the web switch used in previous solutions). In general, this router or switch can use simple load-balancing algorithm such as Round Robin, Weighted Round Robin, etc (The similar strategy as L2S) to forward the requests. Thus, it functions to achieve the coarse-grain load balancing performance.

The dispatching is not finished after the first step. In our approach, the server who receives the request is not always the server who handles it. The second step is used to implement the load-balancing and content-sensitive strategies. In this step, after a server receives the request, it will decide to serve this request by itself or forward it to another server based on the application layer content of the request. Actually, it will perform the content-aware dispatching. Unlike in L2S, here, a distributed hash table is consulted to determine which server has the highest probability to cache the requested object. Thus, no server needs to be aware of the contents on other server's buffer caches to make the dispatching decision.

Due to the heavy-tailed distribution of web requests, on each server, we divide the buffer cache space into two parts: 1) Local Cache, this portion is used to cache the files (or objects) with the highest access frequencies; 2) Global Cache, this piece of cache is used to cache other files based on LRU or GreedyDual algorithm [8]. To calculate the access frequencies of files, a *History Table* is created in the main memory. This table records the access frequencies of the files for the requests which are initially send to the server. There are some other data structures which will be introduced later.

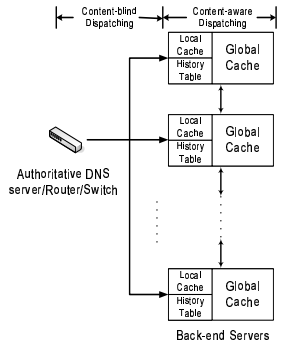


Fig. 2. The System Architecture

### B. Global Cache

By using the content-aware request dispatching, LARD and L2S algorithms could distribute the coming requests to the servers which has the highest probability to cache the requested files. They are able to improve the hit rates in the servers' main memory buffer cache, and reduce the number of slow disk accesses. Thus, a lower average service time per request is obtained. From the point of the view of the outside world, the main memory caches on all the servers are working together as a big virtual cache. In our approach, this is true for global cache only.

In LARD, to better distribute the new coming requests, the front-end switch must keep tracks of the file access history information to aware of the current contents in servers' main memory cache. This imposes significant processing overhead on the front-end and restricts its scalability. L2S uses decentralized control to maintain this information. In L2S, since there's no central facility which has the global view of the memory caches on all the servers, to make the optimal dispatching decision, each back-end server must maintains the up-to-date information on other servers' caches. Thus, each server must broadcast this information to all the other servers periodically or do it in case of the cache contents changing. This message broadcasting requirement introduces significant amount of inter-server messages and consumes high network bandwidth. Although the back-end servers are connected with the high speed local network connections, this problem still could become severe as more and more servers are used. The web cluster could reach its saturation status and could not improve the throughput anymore. Clearly, LARD and L2S can not achieve the optimal performance.

To relieve this problem, we propose a new distributed hash based dispatching algorithm. It is decentralized but does not introduce as much communication overhead as in L2S. In our approach, a collision free hash function

| | | |
|---|---|---|
| Fileid1 | Http link of File1 | status1 |
| Fileid2 | Http link of File2 | status2 |
| Fileid3 | Http link of File3 | status3 |
| ⋮ | ⋮ | ⋮ |
| Fileidk | Http link of Filek | statusk |

Fig. 3. The Structure of Mapping_Table

Zone Table

| Zone | Server |
|---|---|
| [0, 64) | 1 |
| [64, 128) | 2 |
| [128, 192) | 3 |
| [192, 256) | 4 |

Fig. 4. A Sample system with 4 servers, name space 256

(such as SHA-1 [9]) is used to generate a hash value (fileid) for each file in a given name space (for example, $2^n$, n is 64,96 or 128). For each file, the full http link could be used as the seed to generate this id. The generation of these fileids can be done in advance. A file to fileid mapping table (MT), can be created. The structure of this table is shown in Figure 3. The field *status* is used for the status of a file, if this file is not in the current server's cache, status is 0. If it is in the local cache, status is 1, and if it is in the global cache, status is 2. This table is kept in each server's main memory for quick reference. When a new request comes, this table will be searched.

We divide the whole fileid name space into *N* disjointed zones (N is the total number of servers). Each server is assigned a particular zone, and it is only responsible for serving the client requests for the files whose fielids are mapped within its corresponding zone. Here, only global cache is used. A *Zone Table*(ZT) is created on each server maintaining the information of servers and their corresponding zones. Figure 4 shows a simple example. A web cluster contains 4 servers, and the size of the name space is 256. The global cache on server 1 is responsible for caching files whose fileids are between 0 and 64. Consequently, server 2, 3 and 4 are responsible for files with the fileids in zones [64, 128), [128, 192) and [192, 256), respectively.

Under such circumstances, when a server receives a request, it checks the mapping table to see if it has a copy in the buffer cache. If yes, the request will be handled. If not, it checks the ZT table to find out the server who is responsible for this fileid and forwards the request to that server. This approach can be viewed as a variance of the simple static hash mechanism used in parallel processing systems. The difference is that, here, the dispatching decision is made on each individual server only instead of a central control facility. Since there's no namespace overlaps among global caches in different servers, we create a virtual cache which has the maximum size to cache files in the memory and reduce the associated overhead of fetching the files from the slow speed disk storage system.

Our approach differs from L2S significantly in the way of server choosing strategy. We use the hash based function to determine the server that the request to be forwarded while in L2S, each server maintains the main memory information of other servers and make decisions according to that information. Our approach has less overhead since no inter-server communications is necessary.

For the global cache space management, LRU algorithm can be used for content eviction and replacement. Thus, the files with the highest access frequencies will be always kept in the cache. Other mechanisms such as Greedy algorithms can also be used to further improve the caching efficiency. [8], [10].

### *C. Local Cache*

In our system, we take a portion of the buffer cache as the local cache on each server. The local cache is used to keep the copies of the highly accessed files locally. The benefits of using the local caches are based on the access patterns of web server documents. As mentioned in several research papers [11], the accesses of the WWW files are heavy-tailed. This represents a high degree of temporal locality. If we can keep copies of these files on each server, we can satisfy the client request immediately. The system throughput can be further improved. To identify those files, we create a History Table (HT) on each server. Each time, as a server receives a request, it checks the table and modifies the access frequencies of that file. If it is a new file which does not accessed before, a new entry is created. If this file is cached in the local cache, the current server can fulfill the request immediately and does not have to forward the request to another server. By using this strategy, the number of request forwarding operations can be reduced significantly.

With the usage of the local cache, the methods a request to be handled can be categorized into three types: First, fetching a file from the local cache or the global cache on the server which receives the request initially; Second, fetching a file from the global cache on a server which receives this request from another server; Third, in case of a global cache miss, the file has to be loaded from the disk system. The first one has the smallest access delay. The last one has the highest delay.

Although, by taking part of the main memory buffer as the local cache, the aggregated size of the global caches is smaller than the virtual cache in LARD and L2S, it does not affect the overall cache hit rates too much. With a relatively big number of servers, the aggregated cache size is

```
r: the incoming new request
status: 0, new request 1, forwarding request
FET_VALVE: the valve to send the file to the original server
acs_num: statistic information for a certain file

Request_Dispatching(r,status)
{
    if (!status) /* a new request */
    {
      rid = SHA-1(r)
      Modify History_Table
      acs_num(rid)++
      if (r in local_cache or global_cache)
          /* check mapping table */
      {
        handle request r
        return
      }
      else
      {
        forwarding_server = check_zone_table(rid)
        Request_Dispatching(r,1) /* forward the request */
      }
    else /* the request is forwarded from another server */
    {
      if (r in local_cache or global_cache)
        handle request r directly
    }
      else
      {
        fetch it from the disk or get it from a database server
        add it to the global cache
        if (cache is full)
          modify global cache using LRU or Greedy
          algorithm
      }
      if (acs_num > FET_VALVE)
      {
        send a copy of the file to original server
        add it to the server's local cache
      }
    }
}
```

Fig. 5. Pseudo Code of The Request Dispatching Algorithm

big enough to hold most frequently accessed files. Only a small number of less accessed files can not be cached in the global cache and evicted to the disk due to the size limitation. Furthermore, due the creation of multiple copies of frequently accessed files on different servers, we can achieve higher hit rates (including both local and global caches) for those files. The average access time per request could be improved.

### *D. Request Dispatching Algorithm*

Figure 5 describes the pseudo code of our dispatching algorithm. A dispatcher process running this algorithm is created on each server. After a server receives a request, it checks if this is a request sent by the router (not the request forwarded from another server). If so, the server searches its mapping table, if the file is stored in the local or global cache. In case of a cache hit, the server replies the client with the requested file immediately. In case of a cache miss, it checks the zone table and forwards the request to the server who is responsible for the specific zone that the fileid fallen in. Here, TCP handoff technique (If the fileid maps to the zone on itself, no message forwarding occurs) can be applied. In both cases, it modifies the access history table, the number of accesses to that particular file is increased by 1. If necessary, the file will be copied to its local cache from the disk or from another server's global cache. The status field in the mapping table has to be modified accordingly.

If the server detects the coming request is forwarded by another server, it searches the requested file in its global and local caches (using the mapping table). If there's a hit, it reply to the client directly instead of sending the request back to the initial server. In case of a cache miss, it has to load the file from the disk. Then, send it back to the client. It modifies its history table and keeps a copy in its own global cache. In case the global cache is full, another file has to be evicted using LRU or Greedy algorithm. Furthermore, if the number of accesses of this file exceeds a certain value (FET_VALVE), it also sends an additional copy to the initial server, and the initial server will keep a copy in its local cache for the future accesses. Next time, if another request comes for the same file, no request forwarding is needed. In case of the local cache full, the least frequently accessed files in the local cache will be evicted.

Both TCP splicing [3] [4] and TCP handoff [5] [6] mechanisms can be used for request forwarding purpose. However, TCP splicing is useful for a centralized content-aware approach, it also generates considerable overhead on the central facility. In our case, we use a totally decentralized architecture. Thus, the TCP handoff is the ideal candidate. In our approach, we apply TCP handoff technique.

### *E. Avoiding Load Imbalance*

Avoid any server to be overloaded is very important for the whole web cluster to achieve the optimal performance. In our approach, we use a simple load balancing strategy. We define two valves: T1 and T2. T1 is defined as the reacceptance valve and T2 is define as the reject valve (T1 $<$ t2). During system normal operation time, if the workload on a server reaches T2, it notifies all the other servers to stop request forwarding to it. If another server receives a request which maps to this overloaded server's corresponding zone, it has to fetch the file from its local disk system. The overloaded server will not reaccept requests until its workload becomes less than T1.

This is a simple strategy but it can achieve relative good load balancing performance.

### F. Server Failure Problem

In LARD, the failure of the web switch will result in the whole system out of service. While our approach is totally decentralized, this single point of failure problem is completely eliminated. In case a server failure occurs, only the files cached on this server's main memory are affected. Other servers can notice the failure of the server if they can not get any response from the failed server.

Two strategies can be used to handle this problem. In the first strategy, another server will take the service responsibility for the files fallen into the zone of the failed server, no cache redivision is necessary. In the second strategy, the hash space is re-divided into N-1 zones, and the global cache on each server is responsible for a zone with the size 1/(N-1). Then, the workloads previously taken on the failed server are evenly redistributed on the remaining servers. The whole system performance reduces gracefully.

## IV. Experimental Environment

We conduct trace-driven simulation experiments to evaluate the performance of our algorithm. In our simulations, LARD and L2S algorithms are used as the reference models. We assume for each coming request, in LARD, the layer-7 switch has to spend 0.5ms to check the request head. We omit the transmission delay from the front-end to back-end servers. If a request is satisfied from a server cache buffer, the delay is counted as 0.05ms. If it has to be loaded from the hard disk, the latency is set as 10ms. If it has to be forwarded to another server, the processing delay is set as 0.5ms. Any inter-server exchanging message will result in a 0.1ms delay. (Different set of values could be used, if it is reasonable, the conclusion will not change). We use HBD (hash based Dispatching) to denote our algorithm. For L2S, a message is always forwarded to a server which holds the data in the cache (if it has a copy in any server's cache). In HBD, a message is forwarded based on the fileid.

In our simulations, the number of servers in the cluster varies from 1 to 64. The size of memory buffer on each server is set as 64MB. The ratio of local cache to global cache is 1:3. Two metrics are used to evaluate the performance, the cache hit rates and the average access delay per request. Cache hits are further divided into local cache hits and global cache hits.

We use the real world web logs obtained from the National Laboratory for Applied Network Research (NLANR) to evaluate our approach. The trace data are collected on four different proxy servers between Jul. 8th, 2003 to Jul. 14th, 2003. The detailed information of the traces is shown in Table I. "# Files" is the total number of the unique requested files, "% of Requests" is the ratio between the unique files and the total number of requests and "Infinite Cache" is the total size of the unique files.

## V. Performance Evaluation

In this section, we present our simulation results, the comparison results and performance analysis.

### A. Average Service Delay

In the first set of experiments, we evaluate the average service delay for a single request. The trace used in this experiment is uc. The result is shown in Figure 6.

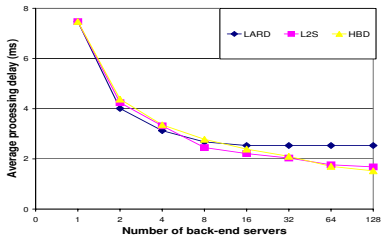


Fig. 6. Average Service Delay Comparison

From the result, we can see, as more servers are introduced in the web cluster, all three algorithms can serve client requests more quickly. When the number of servers is small (2 or 4), LARD outperforms both L2S and HBD. It has the lowest service delay. This is because with the small number of servers, the front-end layer-7 switch used in LARD can handle all the incoming requests, and it can select the best server to satisfy each request. There's no content-blind problem and no inter-server forwarding is necessary. L2S also has better performance than HBD, this is because in L2S, each server tracks the cache content of other servers, if a requested file has a copy in a server's cache, it can always forward the request to that server. However, this is achieved based on the inter-server communications. Since the number of servers is not big, this overhead is affordable. In HBD, we have to send the files blindly, we only check the fileid before the forwarding process, and we have no idea if the server who is responsible for this file has a copy in the cache or not. Though it may results in a little bit lower cache hit rates, we avoid the expensive inter-server message transmission, and we also avoid the potential bottleneck layer-7 switch.

As more servers are added, LARD quickly reach its saturation, as shown in the results, when the number of servers increased from 16 to 32, or even 64, no further improvement we can get in LARD. The reason is the layer-7 switch can not handle more requests, although the back-end servers might idle, there's no way to add more

TABLE I
NLANR WORKLOAD TRACES CHARACTERISTICS

| Traces | # Clients | # Requests | # Files | % of Requests | Total Size | Infinite Cache |
|---|---|---|---|---|---|---|
| bo1 | 551 | 1587478 | 958152 | 60.36% | 19.23GB | 12.61GB |
| pa | 707 | 1551475 | 969639 | 62.50% | 19.68GB | 9.17GB |
| sd | 810 | 2790697 | 1565275 | 56.09% | 31.15GB | 17.74GB |
| uc | 1001 | 4526041 | 1978905 | 43.72% | 44.53GB | 30.07GB |

workloads on these servers. For both L2S and HBD, the server delay per request keep dropping as more servers are added. Although L2S has the best performance, again, as we mentioned earlier, it will result in more inter-server message exchanges. It will cause L2S to reach its saturation very soon. As we can see from the result, when the server number increased from 64 to 128, no performance gain we can obtain in L2S. For our HBD algorithm, this does not happen. Clearly, for large-scale web clusters, our approach is the best.

### *B. Cache Hit Rates Comparison*

In this experiment, we test the main memory cache hit rates in HBD. The result is shown in Figure 7, it includes both local cache and global cache hits.

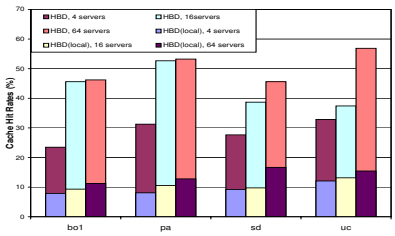


Fig. 7. Cache Hit Rates in HBD, different traces

From the result, we can observe the following phenomenon: First, as more servers are used, the cache hit rates increases. However, when we increase the number from 4 to 16, we can obtain a big improvement. This is because the addition of more cache space make the system can hold more frequently accessed files. When the number increased from 16 to 64, the improvement is not significant. The reason is with 16 servers, we can hold almost all the highly accessed files, add more servers make the system can hold more files with small access frequencies. It does not help too much on the overall hit rates.

Second, for local caches, there's no big improvement as we increase the number of servers. The reason is as more servers are used, the requests are distributed more diverse to more servers. The benefits of using local cache diminishes, but still we can get almost 10% local cache hit rates, which proves the local cache is important. Third, for traces with the different characteristics, the performance will be different. For those traces which have more accesses to the popular files (such as us), HBD performs better. Since most web services have the heavy-tailed distribution, we can predict HBD to achieve very good performance.

## VI. RELATED WORK

There is no doubt that the web cluster is the dominant architecture for scalable web-base applications nowadays. In recent years, many efforts have been devoted on designing a scalable request dispatching algorithm. In [12], V. Cardellini et. al summarizes the most popular dispatching algorithms.

The content-blind algorithms have the advantages of the simplicity and easy implementation, they are widely used in the early times. The typical dispatching algorithms are Random, Round Robin (RR) and Static Weighted Round Robin (WRR). Client Partitioning is another algorithm which providing a simple method to statically partition the server nodes and to assign groups of clients accordingly. These algorithms only consider workload status. While other algorithms such as Least Load First [13], [14], Weighted Round Robin [15] and Client affinity [16] can achieve better performance by taking both the client and the server state information for dispatching decision. Although it is not very efficient, the low dispatching overhead of content-blind algorithms make a single layer-4 switch can support a large number of back-end servers.

Content-aware dispatching algorithms can achieve better performance than the content-blind approach since they can determine which back-end server to be used to satisfy the current request based on the request content. Service partitioning [17] employs specialized servers for certain types of requests. Size Interval Task Assignment with Equal Load (SITA-E) [18] and Client-Aware Policy (CAP) [19] aims to improve load sharing among the servers only. These algorithms do not take server states into account. While, other popular content-aware dispatching algorithms include LARD [1] and HACC [2] consider both client and server states can achieve better performance.

A lot of research papers have been published on the workload characterization of web systems. In [11], the

authors found out that the accesses to the web servers have a heavy-tailed distribution. A small amount of files have much higher access frequency than other files. Our approach takes this into account by creating a local cache on each server to cache the hottest file, the expensive request forwarding operations among different back-end servers are significantly reduced.

In recent years, Peer-to-Peer (P2P) file sharing systems became more and more popular on the Internet. Most P2P applications take the fully distributed mechanism, there's no central server which maintaining the file location information. Thus, with a huge number of peers, how to find the location of a certain object is the main design issue. Chord [20], [21], Pastry [22], Tapestry [23] and CAN [6] use *Distributed Hash Table* (DHT) based routing algorithm for this purpose. In DHT, each peer and each file is assigned a nodeid or fileid using a collision free hash function. The whole namespace is divided into disjointed zone with each peer is responsible for maintaining the location information of the files which fileids are mapped to the corresponding zones. In [24], we developed an efficient web caching system on the client side using P2P technique.

## VII. Conclusions

In this paper, we introduce a new request dispatching algorithm for content-aware web clusters. In our system, distributed hash table is used. We create a fixed mapping relation between a file and the server which responsible for serving this file. On each server, a local cache and a global cache are created to deal with the files with different access frequencies. Our algorithm is simple, efficient and completely decentralized. It solves the existing scalable problems in current Layer-7 dispatching algorithms, and avoids the high inter-server communication overhead in previous solutions. Furthermore, it has better fault tolerant property than both content-blind and content-aware approaches, a server failure will not cause the shutdown of the service. Our simulation results show our approach can significantly improve the system overall performance.

## References


[1] V. S. Pai, M. Aron, G. Banga, M. Svendsen, P. Druschel, W. Zwaenepoel, and E. Nahum, "Locality-aware request distribution in cluster-based network servers," in *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*, (San Jose, CA), October 1998.

[2] X. Zhang, M. Barrientos, J. B. Chen, and M. Seltzer, "Hacc: An architecture for cluster-based web servers," in *Proceedings of the 3rd USENIX Windows NT Symposium*, (Seattle, WA), July 1999.

[3] M. Aron, P. Druschel, and W. Zwaenepoel, "Efficient support for P-HTTP in cluster-based web servers," in *Proceedings of the USENIX Annual Technical Conference*, (Monterey, CA), pp. 185–198, June 1999.

[4] A. Cohen, S. Rangarajan, and J. H. Slye, "On the performance of TCP splicing for URL-aware redirection," in *USENIX Symposium on Internet Technologies and Systems*, 1999.

[5] G. Apostolopoulos, V. Peris, P. Pradhan, and D. Saha, "Securing electronic commerce: reducing the SSL overhead," *IEEE Network*, vol. 14, pp. 8–16, July 2000.

[6] M. Aron, D. Sanders, P. Druschel, and W. Zwaenepoel, "Scalable content-aware request distribution in cluster-based network servers," in *2000 USENIX Annual Technical Conference: San Diego, CA, USA, June 18–23, 2000* (USENIX, ed.), (Berkeley, CA, USA), pp. 323–336, USENIX, 2000.

[7] E. V. Carrera and R. Bianchini, "Efficiency vs. portability in cluster-based network servers," *ACM SIGPLAN Notices*, vol. 36, pp. 113–122, July 2001.

[8] S. Jin and A. Bestavros, "Popularity-aware greedydual-size algorithms for Web access," in *Proceedings of the 20th International Conference on Distributed Computing Systems (ICDCS)*, (Taipei, Taiwan, ROC), Apr. 2000.

[9] F. I. P. S. Publication, "Secure hash standard, http://www.itl.nist.gov/fipspubs/fip180-1.htm," 1995.

[10] S. Jin and A. Bestavros, "GreedyDual* Web caching algorithms: Exploiting the two sources of temporal locality in Web request streams," in *Proceedings of the 5th International Web Caching and Content Delivery Workshop*, (Lisbon, Portugal), May 2000.

[11] M. Crovella and A. Bestavros, "Self-Similarity in World Wide Web Traffic: Evidence and Possible Causes," in *Proceedings of SIGMETRICS'96: The ACM International Conference on Measurement and Modeling of Computer Systems.*, (Philadelphia, Pennsylvania), May 1996. Also, in Performance evaluation review, May 1996, 24(1):160-169.

[12] V. Cardellini, E. Casalicchio, M. Colajanni, and P. S. Yu, "The State of the Art in Locally Distributed Web-server Systems," Technical Report, IBM T.J. Watson Research Center, 2001.

[13] Cisco Systems, "Local Director, http://www.cisco.com."

[14] F5 Networks, "BIG/ip, http://www.f5labs.com."

[15] G. D. Hunt, G. S. Goldszmidt, R. P. King, and R. Mukherjee, "Network dispatcher: A connection router for scalable internet services," *Computer Networks*, vol. 30, no. 1-7, pp. 347–357, 1998.

[16] Linux Virtual Server Project, "http://www.linuxvirtualserver.org."

[17] C. Yang and M. Luo, "A content placement and management system for distributed web-server systems," in *Proceedings of 20 International Conference on Distributed Computing Systems*, pp. 691–698, 2000.

[18] M. Harchol-Balter, M. E. Crovella, and C. D. Murta, "On choosing a task assignment policy for a distributed server system," *Journal of Parallel and Distributed Computing*, vol. 59, no. 2, pp. 204–228, 1999.

[19] E. Casalicchio and M. Colajanni, "A client-aware dispatching algorithm for web clusters providing multiple services," in *World Wide Web*, pp. 535–544, 2001.

[20] I. Stoica, R. Morris, D. Karger, M. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for Internet applications." Technical Report TR-819, MIT., Mar. 2001.

[21] F. Dabek, E. Brunskill, M. F. Kaashoek, D. Karger, R. Morris, I. Stoica, and H. Balakrishnan, "Building Peer-to-Peer Systems With Chord, a Distributed Lookup Service," in *the 8th IEEE Workshop on Hot Topics in Operating Systems (HotOS), Schoss Elmau, Germany*, pp. 195–206, May 2001.

[22] A. I. T. Rowstron and P. Druschel, "Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems," in *Proceedings of the 18th IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, (Heidelberg, Germany), pp. 329–350, Nov. 2001.

[23] B. Zhao, J. Kubiatowicz, and A. Joseph, "Tapestry: An infrastructure for fault-tolerant widearea location and routing." Technical Report UCB/CSD-01-1141, U.C.Berkeley, CA, 2001.

[24] Z. Xu, Y. Hu, and L. Bhuyan, "Exploiting client cache: A scalable and efficient approach to build large web cache," in *Proceedings of International Parallel and Distributed Processing Symposium, (IPDPS'04)*, (Santa Fe, NM), Apr 2004.