IPSJ Transactions on Advanced Computing Systems Vol. 4 No. 2 59–69 (Mar. 2011)

Regular Paper

DTS: Broadcast-based Content-aware TCP Connection Handover

Hajime Fujita[†1] and Yutaka Ishikawa[†1,†2]

In this paper we propose DTS (Distributed TCP Splicing), a new mechanism for performing content-aware TCP connection switching in a broadcast-based single IP address cluster. Broadcast-based design enables each cluster node to continue to provide services to clients even when other nodes in the cluster fail. Each connection request from a client is first distributed among the cluster using the consistent hashing method, in order to share the request inspection workload. Then the connection is transferred to an appropriate node according to the content of the request. DTS is implemented on the Linux kernel module and does not require any modification to the main kernel code, server applications, or client applications. With an 8-node server configuration, a DTS cluster with multiple request inspectors achieves about 3.4 times higher connection throughput compared to the single inspector configuration. A SPECweb 2005 Support benchmark is also conducted with a four node cluster, where DTS reduces the total amount of disk accesses with a locality-aware request distribution and almost halves the number of file downloads that fail to meet the speed requirement.

1. Introduction

The Internet and servers on the net have never been more important, as many services have shifted to web services, sometimes on clouds. Needless to say there has been a consistent demand for servers to be more powerful and reliable.

Single IP address clusters [1] have been widely adopted in servers for several reasons. One reason is for performance improvement, as server workloads are nearly independent of each other and easily parallelized to multiple computers. Another reason is for reliability, to add redundancy to the server system.

Existing single IP address clusters are classified based on several aspects. One aspect is how a cluster distributes the input from clients to member nodes. In general, there are two types, the dispatcher-based type and the broadcast-based type. The dispatcher-based type has one special node called a "dispatcher" or "load balancer". This node has an IP address that represents the cluster, and receives all incoming packets from the clients and forwards them to other nodes in the cluster. Examples of this type of cluster system include Linux Virtual Server [2),3)] and SAPS [4)]. On the other hand, the broadcast-based type does not have a special node, but all nodes have the same IP address which represents the cluster. The front-side switch or router is configured so that all incoming packets from the client are broadcast to all server nodes. Examples of this type are the Network Load Balancing feature [5)] of Windows Server families and FTCS [6)].

Another aspect used to classify single IP address clusters is whether it handles the content of a client's request to choose the appropriate node to process the request. For example, in a Web server there might be a file which is only located on specific nodes. In this case, requests for this file should be processed at nodes which have the file in local storage. Another example of the application of content-aware request distribution is Locality Aware Request Distribution [7),8)], which assigns requests for the same content to the same node to achieve a higher disk-cache hit rate. This kind of request distribution is called content-aware request distribution. It is also sometimes called layer-7 request distribution, since it considers the layer-7 protocols in the OSI reference model.

In order to support content-aware request distribution for TCP-based applications, a special mechanism is required. This is because a new connection must be established between a client and one of the cluster nodes in order to receive the content of the request for inspection, but the final destination of the connection might be different from the node which inspects the request. To improve the performance of content-aware request distribution, several kernel-level methods like TCP splicing [9)] or TCP handoff [10)] and their variants were proposed, but most of them are based on dispatcher-based clusters.

In this paper we propose DTS (Distributed TCP Splicing), which is a mechanism for constructing broadcast-based single IP address clusters with content-aware request dispatching. We reported on the TCP connection handover mechanism of DTS in Ref. 11). Following on from the previous paper, this paper

†1 Information Technology Center, the University of Tokyo
†2 Graduate School of Information Science and Technology, the University of Tokyo

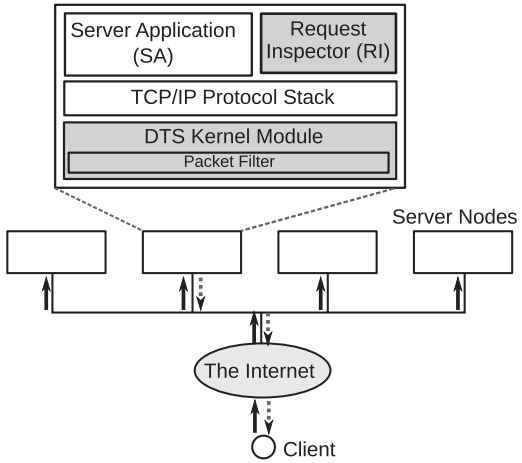

Fig. 1 Overview of DTS.

describes a layer-4 load balancing feature for request inspection and performance evaluation. Incoming connections from clients are first distributed among the cluster nodes by using a layer-4 load balancing feature based on the consistent hashing [12]. DTS was implemented on the Linux kernel and connection handling throughput has been measured. With 8-node server nodes, the multiple request inspector configuration achieves about 3.4 times higher throughput compared to the single inspector configuration. Additionally, DTS enables employment of a locality-aware request distribution and reduces disk accesses in the SPECweb2005 Support benchmark.

2. Design

In this section the design of DTS is described. DTS is a content-aware request dispatching scheme for TCP-based server applications on broadcast-based single IP address clusters.

Figure 1 shows a typical configuration of a DTS-based four-node cluster. The cluster is configured as a broadcast-based cluster, that is, all incoming packets are sprayed across all server nodes. Each server node has two IP addresses. One is for communicating with remote clients, and all the nodes in the cluster share this address. The other one is for communicating with other server nodes, which is unique to each node.

Each server has the same software stack configuration also shown in Fig. 1. Two software components, **"DTS Kernel Module"** and **"Request Inspector"** are DTS-specific components added into commodity operating systems. Inside the kernel module there is a packet filter which captures and modifies the packet coming in to or going out from a server node. As its name suggests, the request inspector is a helper application program which receives a request from a client and decides which server node should take care of the request.

The key point of the DTS design is that the new features are realized by just adding those two components into existing operating systems, like Linux. Not even the TCP/IP stack needs to be modified.

2.1 Accepting Requests

When a client attempts to make a new TCP connection with a DTS-based cluster, the connection is accepted by only one node in the cluster, called the **request inspecting node** (the node at the left side of **Fig. 2** (a)). Actually in DTS, any server node can be a request inspecting node, because all nodes are equivalent in terms of network topology and all nodes have the same software configuration. For each incoming connection, the request inspecting node is chosen by a hash-based layer-4 load balancing scheme described later, in Section 2.3. Packets from the client also flow to other nodes, but they are blocked by the packet filter.

On the request inspecting node, the connection is first handled by the request inspector to analyze the contents of the request. **Figure 3** shows the pseudo-code of the request inspector. It is almost the same as a socket-based server program, except that it issues a DTS-specific system call called `request_handover` to request the kernel module to transfer the connection to another node.

Upon receiving the request, the inspector determines the most appropriate node to handle it, which is called the **request processing node**, and issues the `request_handover` system call to transfer the connection to the chosen node (Fig. 2 (b)). In the system call routine, the kernel module on the request inspecting node first creates a new TCP connection to the server application on the request processing node, then transmits the request passed from the inspector. This new connection is adjusted by the packet filter in the kernel module so that

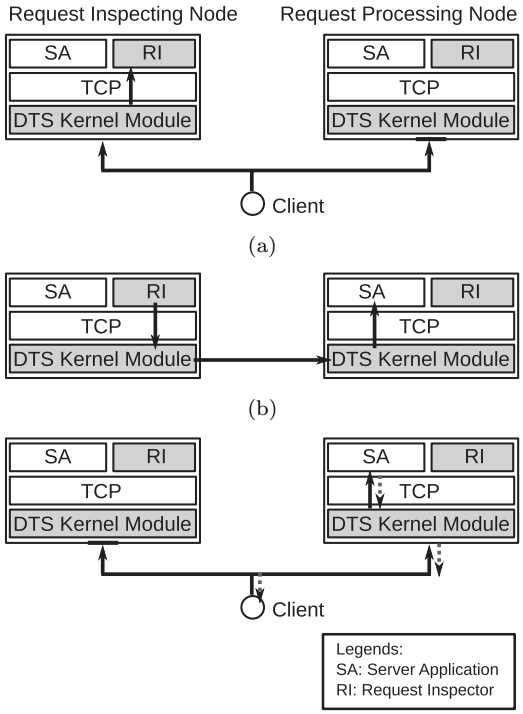

Fig. 2 Overview of the Handover procedure.

the request processing node sees the connection as if it came from the client directly. After receiving the request from the inspecting node, the processing node begins to communicate with the client (Fig. 2 (c)). At this moment two connections, one is the connection between the client and the server, the other is between the sender and the receiver, are *splice*d and can be seen as a single TCP connection. This handover process is completely carried out in the server side kernel module, so clients and server applications do not need to be modified.

After the handover process is completed, the client and the request processing node communicate with each other directly. In other words, the request inspecting node is no longer involved in this communication. This is because incoming

```
while (1) {
    int s;
    char buf[BUFSIZ];
    ssize_t ret;

    s = accept(...);

    ret = read(s, buf, sizeof(buf));

    /* parse the request and determine the target node */
    ...;

    request_handover(s, target, buff, ret);
}
```

Fig. 3 The pseudo-code of the request inspector.

packets from clients are delivered to each node without being forwarded by another node.

2.2 Packet Sequence of the Handover Process

A more precise example of connection handover procedure is shown in **Fig. 4**. One client and two server nodes are drawn in the figure.

First, the client sends a SYN segment to the server (Segment 1 in Fig. 4). Suppose that server node 1 is the request inspection node for this connection, so the connection is accepted by node 1. Actually the same packet reaches node 2 and is eventually dropped by the packet filter, but it is omitted in the figure for simplicity. At node 1, the destination port number in the packet is modified so that it is delivered to the request inspector, not to the server application. On the other hand, when the packet is transmitted from node 1, the source port number is set to that of the server application.

When the inspector receives the request and determines the request processing node, it issues the `request_handover` system call to transfer the connection. Suppose that the chosen node is server node 2 in this example. In the system call, the kernel module opens a new TCP connection to the request processing node (Segment 4). Since the initial sequence number (ISN) of each TCP connection is generated randomly, the ISN of this new connection is different from that of segment 1. The packet filter in the DTS kernel module hooks the packet and

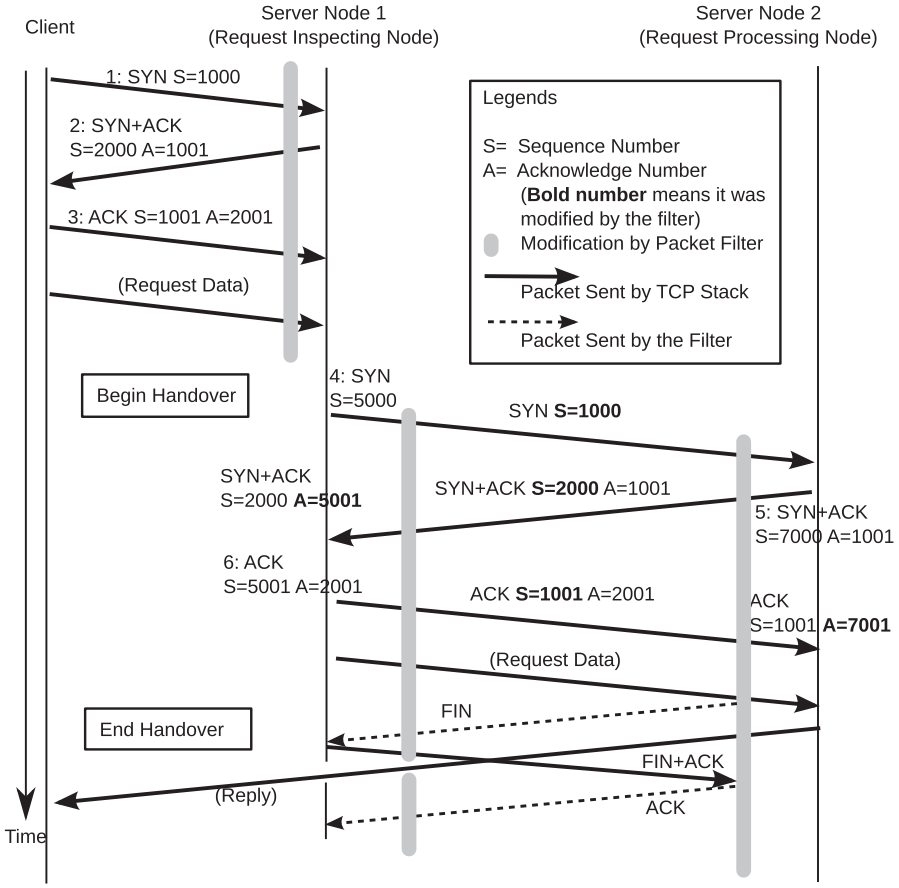

Fig. 4 Packet sequence.

modifies the sequence number so that it is equal to that of segment 1. When this segment arrives at node 2, the packet filter rewrites the source IP address and the source port number to those of the client. From node 2's TCP stack, it seems as if segment 1 came from the client directly. When node 2 replies to the client with segment 5, the sequence number, addresses, and port numbers are also modified so that the sequence number becomes equal to that of segment 2 and also the segment goes back to node 1 instead of the client. Upon receiving segment 2, the filter at node 1 rewrites the acknowledgment number so that the segment can be accepted by the TCP stack.

When node 2 is about to receive segment 4, the filter in node 2 must know the client's address, port number, and the original ISN. This information is stored in a DTS-specific TCP option at the head of segment 4.

After the reception of the request from node 1 is completed, node 2 begins to send packets to / receive packets from the client. Since the sequence numbers have been adjusted, the TCP protocol stacks of both the client and node 2 do not recognize that the actual communicating peer has been changed. Also from this point node 2 is able to communicate with the client, in other words, it no longer depends on node 1.

When node 2 begins to communicate with the client, the connection between node 1 and node 2 which was created for the handover is no longer needed. Therefore the packet filter in node 2 transmits a FIN segment to clean up the connection. It is important to send a FIN from the request processing node side, otherwise there would be many TIME-WAIT connections in the request inspection side if connection handovers were done at a very high rate. This would make the handover unavailable, because eventually all ephemeral ports would be consumed.

2.3 Layer-4 Load Balancing

In order to achieve content-aware request dispatching, requests from clients should first be inspected by an inspector. This is a somewhat heavy task compared to content-blind connection dispatching, i.e., layer-4 dispatching. In DTS, the request inspecting node is determined by a layer-4 load balancing method using the consistent hashing [12] scheme.

Each node has a unique hash value based on a unique ID like an IP address for intra-cluster communications. Also each incoming connection has a hash value based on a remote address and a port number. These hash values are mapped to a conceptual ring which represents the range of the hash function (**Fig. 5**). The ring is separated into several areas by the hash values of the nodes. Each connection falls into one of the areas. A connection is accepted by a node i when the following condition is met for any node $j \neq i$,

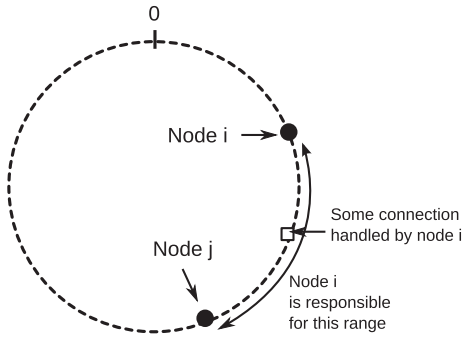

Fig. 5 Hash value space.

$h(n_i) \leq h(a, p) < h(n_j)$

where n_i and n_j are unique IDs for nodes i and j, a is a remote address, p is a remote port number, and h is a hash function. While it is shown that a hash-based static connection distribution is not sufficient for load balancing [6], we do not need to concern ourselves with it because in DTS we have another chance to distribute the workload among cluster nodes with a connection handover.

Just using a single ID for each node does not ensure that the distance between each hash value is equal. This leads to a load imbalance for request inspection. Therefore we use a virtual node technique like the one used in Amazon's Dynamo [13]. With this technique, each (physical) node has multiple hash values on the hash space ring. This is just as if each physical node hosts multiple "virtual nodes" on the ring.

We assume that every node in the cluster has a knowledge of what nodes are currently in the cluster and what hash values they have.

2.4 Connection Tracking

As described above, DTS handles several TCP connections simultaneously especially during the handover process. For example, at the request processing node there are two TCP connections, one is between the inspecting node and the processing node and the other is between the processing node and the client.

To track the state of each connection, DTS has its own connection table. When the packet filter sees a packet, it first looks it up in the connection table by using a quadruple of remote address, remote port number, local address, and local port number as a key. If an entry is not found the packet is regarded as being handled by another node and just ignored.

2.5 Server Failures

The design of DTS assumes that a server node may fail without any symptoms. We do not provide full fault-tolerance. Our goal is to keep the cluster available even when some of the nodes die. If a node crashes, connections served by that node are lost. Clients that lose connections must reconnect to the server.

As in broadcast-based clusters, a server node in DTS is not affected by other node failures, because each node is able to exchange packets with clients without depending on any other node. While it is possible to make a redundant configuration for dispatchers in the dispatcher-based type clusters, there would be a downtime after a dispatcher's crash until a backup dispatcher starts working. During this period, all clients have to wait for the new dispatcher to send packets to server nodes behind the dispatcher. On the other hand, with broadcast-based clusters, including DTS, most clients do not observe this kind of downtime except for the clients that were connected to the crashed node.

Having multiple request inspectors in a cluster also improves user experiences under server failures. When a client tries to establish a new TCP connection with a server, a server node might crash, but the crashed node might not be removed from the consistent hash space. In such a situation, if we assume the ideal case that consistent hashing chooses one of the server nodes in a perfectly random manner, the probability of a client trying to connect to the crashed node goes down as the number of request inspecting nodes increases.

Consistent hashing is also favorable in its handling of server failures. If a server node fails, the adjacent node in the hash space takes over the area covered by the crashed node. This does not affect a node whose area is not adjacent to that of the crashed node. Existing broadcast-based clusters use a hash function that depends on the total number of cluster nodes, therefore every node should move in the hash value space.

DTS itself does not solve the cluster membership problem. It can be implemented using existing cluster management software like Heartbeat [14].

2.6 Applicability

Since DTS is designed to interact with application-level information, it includes

some assumptions on application-level protocols. The current design assumes that a client first sends a request to a server, as in HTTP [15]. This assumption may be relaxed for protocols in which a server sends messages before a client sends a request, given that the messages sent by the server are predictable. For some protocols where messages are not predictable, for example, protocols that perform challenge and response authentications, DTS can never be applicable.

3. Implementation

We implement DTS on the Linux kernel. DTS is implemented as a packet filter using the Netfilter [16] mechanism. DTS is wholly implemented as a loadable kernel module, therefore no modification (patching) to the main kernel source code is needed.

3.1 Consistent Hashing

The current implementation of the consistent hashing algorithm uses the MD5 digest function to calculate hash values. Each server node has its own IP address (for intra-cluster communications) and a set of small integer numbers. A hash value for a virtual node is calculated by just concatenating the IP address and an integer number and applying MD5 to it. Also connections from clients are handled in a similar way. The IP address and the port number of a client is concatenated and passed to the hash function.

3.2 Implementation Limitations

The current implementation of DTS has several limitations related to advanced features of TCP. For example, Selective ACK, TCP timestamp, and Window scaling are not supported. These features must be disabled at the server side. Since these features are used only when both peers of the connection are willing to use them, we can safely ignore them. the current implementation does not consider IP fragmentation either. However this is not a fatal issue because many TCP implementations set the DF (don't fragment) bit to IP packets mainly for performance reasons.

These limitations are just implementation limitations, not design limitations. The authors believe that it is possible for DTS to support these features. When DTS is used in a real Internet environment, these features should be supported because these TCP options improve the throughput in long-latency networks.

Table 1 Machines used for the experiments.

Server	
CPU	Intel Xeon L5520 (2.26 GHz, 4 cores) (Hyper-Threading and Turbo Boost enabled)
Memory	3 GB
NIC	Broadcom NetXtreme II BCM5716 x 2
OS	Ubuntu Linux 10.04 amd64
OS Kernel	Linux 2.6.32-23-generic (x86_64)
Web/JSP server	Tomcat 6.0.24
Java Runtime	OpenJDK 1.6.0_18
Client A × 5	
CPU	AMD Opteron 175 (2.2 GHz, 2 cores)
Memory	2 GB
NIC	Broadcom NetXtreme BCM5721
OS	Ubuntu Linux 8.04 i686
OS Kernel	Linux 2.6.24-28-generic (i686)
Client B × 1	
CPU	Intel Xeon X5520 (2.67 GHz, 4 cores × 2) (Hyper-Threading and Turbo Boost disabled)
Memory	12 GB
NIC	Broadcom NetXtreme II BCM5716
OS	Ubuntu Linux 10.04 amd64
OS Kernel	Linux 2.6.32-23-generic (x86_64)
Switch	
Model	DELL PowerConnect 6248

4. Evaluation

In this section, the basic performance of DTS is shown first, then the results of a web application benchmark are shown.

4.1 Environment

In the experiments below, the machines shown in **Table 1** are used. All the server nodes and the client machines are connected to the same switch. Two VLANs are configured, one is for client-server communication and the other is for intra-cluster communication.

4.2 Scalability

The scalability of DTS is measured. In this experiment a simple ping-pong echo server and client are used as a workload.

The server cluster is configured in two ways. One configuration is a **single request inspector (single RI)** configuration, where only one server node hosts

a request inspector. This configuration does not use layer-4 load balancing with consistent hashing. The other configuration is a **multiple request inspectors (multiple RIs)** configuration, where each node hosts a request inspector and new connections from clients are first distributed among nodes by layer-4 load balancing. Each server node has 63 virtual nodes in the hash space.

In this experiment, the inspector virtually does nothing; it just receives a predefined size of data from the client and immediately requests a handover of the connection to one of the nodes in the cluster, chosen in a round-robin manner, including the node on which the inspector runs. This inspector setting shows the minimum overhead when using DTS.

As a server application, a simple echo back server is used. A client first sends data, then a request inspector receives the entire segment of data and passes it for handover. After that, the server program receives the data and sends it back to the client.

Two data sizes, 4 bytes and 1024 bytes, are used. 1024 bytes of data is close to the typical size of HTTP headers which include cookies. For example, nine requests were observed when displaying `www.google.com` with Firefox 3.5.9 and the average of the length of each HTTP request was 883 bytes.

Six client machines try to connect to the server simultaneously for 30 seconds and the total number of processed requests is measured.

Figures 6 and **7** show the results. Generally, multiple request inspector configurations achieve a higher performance. With a single inspector, the throughput does not improve, even when the total number of server nodes increases. From this result, we see the single inspector is a bottleneck in this case. By using multiple inspectors we can scale up the cluster. In Fig. 7, the multiple inspector case saturates at around 85,000 connection/s. In this case each connection transmits 1 KB of data, so 85,000 connections per second means 85 MB/s. In this case, it is likely that the network bandwidth (1 Gbps) is almost saturated.

4.3 Virtual Nodes

The number of virtual nodes affects the load balance among request inspecting nodes, so the throughput measurement, as described above, is conducted while changing the number of virtual nodes. The configuration is the same as the "8-node cluster with multiple request inspectors" setting above, except that the number of virtual nodes per each server node is altered.

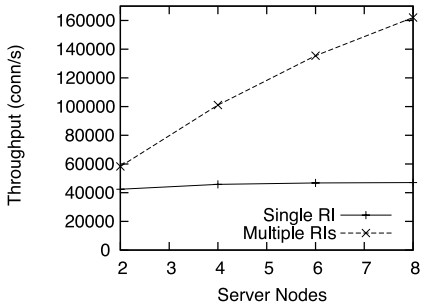

Fig. 6 Throughput scaling (4 bytes).

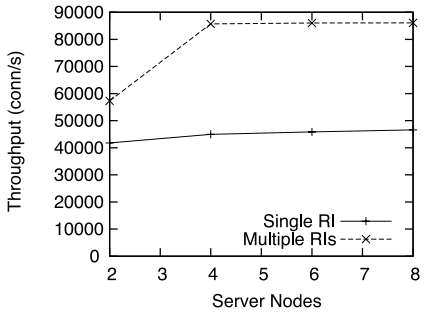

Fig. 7 Throughput scaling (1,024 bytes).

Figure 8 shows the results. In the figure, "min" represents the percentage of the connections processed by the node with the minimum load. Also, "max" represents the percentage of the connections processed by the most loaded node. If the difference between "min" and "max" is greater, the load is imbalanced. From the figure, with at least 31 virtual nodes per physical node, a total of 248 virtual nodes are required to balance the request inspection workload almost equally.

4.4 Web Application Benchmark

A Web application benchmark test is conducted in order to show that context-aware request dispatching is useful for real applications.

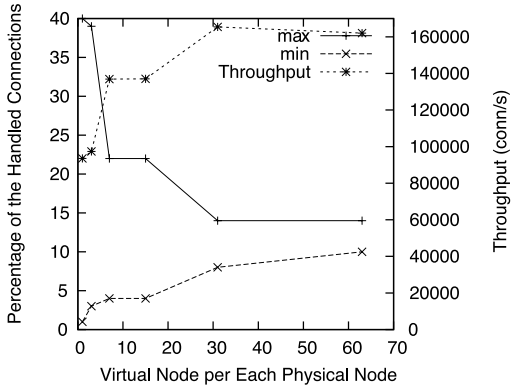

Fig. 8 Number of virtual nodes and overall performance.

Table 2 Parameters changed from the default in the support benchmark.

Parameter	Value used	Default
Simultaneous Sessions	1,800	N/A
Benchmark Run Seconds	900	1,800
Benchmark Iterations	1	3
Download Speed (bytes/s)	1,000,000	100,000
for "Good" QoS	990,000	99,000
for "Tolerable" QoS	950,000	95,000

The "Support" application which comes with the SPECweb2005 benchmark suite [17] is used as a benchmark. This benchmark simulates a user support web site where users search and download device drivers or software updates for various products. The web site consists of dynamic pages generated by JSP (Java Server Pages), and a huge set of static download files.

Four server nodes and five client nodes (client A in Table 1) are used for the experiment. Every server node has the same file set so that each node can handle any request. Apache Tomcat [18] is used as both the Web server and the JSP server. This means both dynamic JSP pages and static contents are processed by Tomcat. Tomcat is configured so as not to use persistent connections, because currently DTS does not support inspecting multiple requests on a single TCP connection.

Table 2 shows adjusted parameters. By default, the download speed per each client is limited to 100 KB/s. This is changed to 1 MB/s to put more pressure on disks and to simulate modern clients over the Internet. QoS criteria for download speeds are also changed. According to the rules of the SPEC benchmark, these values other than Simultaneous Sessions, are not allowed to be modified, therefore results shown here are not valid SPEC scores and thus not comparable to other published results. However, it is enough to see performance characteristics under different server configurations.

```
   uri = read_http_request_uri();

A: if (!contains(uri, "/downloads/")
B:   || contains(uri, "dir000000000")) {
       target = myself;
   }
   else {
       hash = md5(uri);
C:     target = node_from_consistent_hash(hash);
   }

   request_handover(target);
```

Fig. 9 The pseudo-code of the L7 request dispatching algorithm.

4.4.1 Request Dispatching Method

In the experiment, we use two server configurations, L4 (layer-4) load balancing and L7 (layer-7; content-aware) load balancing. In both configurations, a new connection from a client is accepted by one of the server nodes, chosen by consistent hashing. Each server node has 63 virtual nodes in a hash space. In the L7 configuration, a request inspector first accepts the connection to determine a request processing node. On the other hand, in the L4 configuration, Tomcat accepts the connection directly instead of the request inspector. This means that in the L4 configuration, a request processing node is determined only by the use of consistent hashing.

The L7 configuration is almost the same as the L4 configuration for most files, but it also takes file locality into consideration for some files. In the L7 configuration, the request inspector uses the algorithm shown in **Fig. 9**. This algorithm is aimed at effectively using the disk cache of each server node when

Table 3 Results of the SPECweb2005 support benchmark tests.

Method	File Type	Total Requests	QoS Good	QoS Tolerable	QoS Fail
L7	Download	21,040	20,879	109	52
	JSP	288,493	288,112	378	3
L4	Download	20,859	20,643	116	100
	JSP	286,878	284,938	1,935	5

downloading large files, which are stored under the "downloads" directory. A basic purpose of this algorithm is to determine a request processing node for a large download file so that the same file is fetched from the same node. The condition in line A is used to handle dynamic scripts and small images on the same node as that chosen by the layer-4 load balancing. Line C determines a request processing node. First an MD5 digest of the requested URI is calculated, and the request processing node is chosen by consistent hashing using the digest. As in the hash space for layer-4 balancing, each server node has 63 virtual nodes in this space.

The condition in line B includes a trick for improving performance. In the Support benchmark, the probability of each file being requested is not equal. Download files are separated in directories "dir0000000001", "dir0000000002", and so on. Files are more likely to be requested if the number in the directory name is small. By exploiting this knowledge, the algorithm lets each node handle requests for files in "dir0000000001" – "dir0000000009", which is likely to be stored in the disk cache of each node.

The request inspector used here is written in the C language and uses the RFC1321-based MD5 library [19].

4.4.2 Results

Table 3 shows the results of the Support benchmark tests. For both server configurations, results for download files are important here because we treat these files specially in the L7 configuration. QoS has different meanings for downloads and JSPs. For download files, QoS stands for download speed described in Table 2. On the other hand, QoS for JSPs is a response time for each request. Responses in less than 3 seconds are classified as "Good" and ones in less than 5 seconds are classified as "Tolerable". For both file types, "Fail" means the number of requests that failed the criteria for "Tolerable". From the overall result, the L7 method achieved better results in terms of both total number of requests and QoS. In particular, with the L7 configuration about 1.1% more files were downloaded within the "Tolerable" QoS range, while the number of "Failed" requests is about half of that of the L4 configuration.

Table 4 Disk I/O statistics during benchmark run.

Method	Average Number of Threads Waiting Disk I/O	Average Disk Reads (KB/s)
L7	0.028/0.011/0.061/0.22	2,839/ 3,196/ 7,330/10,169
L4	0.20/0.12/0.21/0.12	11,038/10,294/11,970/11,513

Table 4 shows disk I/O statistics measured during the benchmark run. The four numbers in each cell show the values for each server node, node 0, 1, 2, and 3, respectively. It can clearly be seen that in the L7 configuration, the total amount of disk I/O is far less than that of the L4 configuration. From this result, it can be said that the locality-aware request distribution employed in the L7 configuration increases the disk cache hit rate.

From Table 3 it can also be seen that the L7 configuration achieved a better result for JSPs. This might be a result of reducing disk accesses, because JSP requests and download requests are sent simultaneously to the server, but currently the authors do not have any evidence to clearly explain this result.

5. Related Work

TCP splicing [9] uses a mechanism similar to DTS to splice two TCP connections inside the kernel of the dispatcher node. Sequence numbers and acknowledge numbers are adjusted so that a packet from an endpoint of one connection can be accepted by an endpoint of another connection. TCP handoff [10] modifies some parameters in a TCP control block, like the sequence number to be transmitted (`SND.NXT`). This makes it unnecessary to rewrite packets in the dispatcher. However, this design requires the modification of the core network protocol stack of the kernel, which is sometimes difficult or impossible.

Some other parties already proposed the idea that request inspection for content-aware connection dispatching should be done on each server node to increase scalability. Kerdlapanan, et al., proposed [20] the use of TCP-handoff for

multicast-based single IP address cluster systems, which is very similar to DTS. However in the paper they just introduced the idea but did not show an implementation or performance evaluation. KNITS [21] is a dispatcher-based single IP address cluster, but request inspection is done by a proxy program (equivalent to the request inspector in DTS) on each node. The main difference between DTS and KNITS is that in KNITS, each server node depends on the dispatcher for forwarding packets from the client. Also the sequence number manipulation is done at the dispatcher. This makes the dispatcher a single point of failure for the cluster. Reference 10) also proposed a system similar to KNITS but uses TCP-handoff. This system also requires a single layer-4 switch at the front of the cluster so this could be a single point of failure.

Marwah, et al., [22] addressed the scalability and fault tolerance of TCP splicing. Connection state information in the proxy is replicated on other proxy nodes. With this replication, any proxy node is able to handle any connection, resulting in improved performance and availability. In contrast, DTS does not need to replicate state information to achieve the same goal, as DTS is broadcast-based, and does not need packet forwarding inside a cluster.

6. Conclusions

In this paper we have proposed DTS, a mechanism for constructing a broadcast-based single IP address cluster with content-aware request distribution. The request inspection workload is distributed among all the member nodes by the consistent hashing method to prevent a request inspector from becoming a bottleneck. After request inspection, the connection is transferred to the request processing node. This process is transparent to the client, the server application, and even the TCP protocol implementation inside the server's kernel.

Experimental results show that DTS improves the scalability of the request inspection workload. Results also show that with the SPECweb2005 Support benchmark, content-aware request distribution reduces disk accesses and decreases the number of file downloads that failed to meet the bandwidth requirement by 48%.

Future work is to support multiple handovers [8], which allows a connection to move across server nodes multiple times. The current implementation only handles the content of the first request in a connection.

Acknowledgments The authors would like to thank Kazuki Ohta and Balazs Gerofi for useful discussions about the consistent hashing algorithm, and Taku Shimosawa for helping us with setting up one of the machines used in our experiments. This work has been supported by the CREST project of JST (Japan Science and Technology Agency).

References

1) Cardellini, V., Casalicchio, E., Colajanni, M. and Yu, P.S.: The State of the Art in Locally Distributed Web-Server Systems, *ACM Computing Surveys*, Vol.34, No.2, pp.263–311 (2002).
2) Zhang, W.: Linux Virtual Servers for Scalable Network Services, *Linux Symposium* (2000).
3) O'Rourke, P. and Keefe, M.: Performance Evaluation of Linux Virtual Server, *LISA 2001 15th Systems Administration Conference*, pp.79–92 (2001).
4) Matsuba, H. and Ishikawa, Y.: Single IP address cluster for internet servers, *Proc. 21st IEEE International Parallel and Distributed Processing Symposium* (*IPDPS2007*) (2007).
5) Microsoft Corporation: Network Load Balancing Technical Overview. http://technet.microsoft.com/en-us/library/bb742455.aspx
6) Fujita, H., Matsuba, H. and Ishikawa, Y.: TCP Connection Scheduler in Single IP Cluster, *8th IEEE International Symposium on Cluster Computing and the Grid* (*CCGRID'08*), pp.366–375 (2008).
7) Pai, V.S., Aron, M., Banga, G., Svendsen, M., Druschel, P., Zwaenepoel, W. and Nahum, E.: Locality-Aware Request Distribution in Cluster-based Network Servers, *Proc. 8th ACM Conference on Architectural Support for Programming Languages and Operating Systems* (1998).
8) Aron, M., Druschel, P. and Zwaenepoel, W.: Efficient Support for P-HTTP in Cluster-based Web Servers, *Proc. USENIX 1999 Annual Technical Conference* (1999).
9) Spatscheck, O., Hansen, J.S., Hartman, J.H. and Peterson, L.L.: Optimizing TCP forwarder performance, *IEEE/ACM Transactions on Networking*, Vol.8, pp.146–157 (2000).
10) Aron, M., Sanders, D., Druschel, P. and Zwaenepoel, W.: Scalable Content-aware Request Distribution in Cluster-based Network Servers, *Proc. USENIX 2000 Annual Technical Conference* (2000).
11) Fujita, H. and Ishikawa, Y.: TCP Connection Handover Mechanism for Layer-7 Load Balancing, *IPSJ SIG Notes*, Vol.2009-OS-111, No.1 (2009).
12) Karger, D., Lehman, E., Leighton, T., Panigrahy, R., Levine, M. and Lewin, D.:

Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web, *STOC '97: Proc. 29th Annual ACM Symposium on Theory of Computing*, New York, NY, USA, ACM, pp.654–663 (1997).
13) DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P. and Vogels, W.: Dynamo: Amazon's highly available key-value store, *SOSP '07: Proc. 21st ACM SIGOPS Symposium on Operating Systems Principles*, pp.205–220 (2007).
14) Robertson, A.: Linux-HA heartbeat system design, *ALS'00: Proc. 4th Conference on 4th Annual Linux Show case & Conference, Atlanta*, Berkeley, CA, USA, USENIX Association, pp.305–316 (2000).
15) Fielding, R., Gettys, J., Mogul, J., Frystyk, H. and Berners-Lee, T.: Hypertext Transfer Protocol – HTTP/1.1, RFC 2068 (1997).
16) The Netfilter.org project: netfilter/iptables project homepage. http://www.netfilter.org/
17) Standard Performance Evaluation Corporation: SPECweb2005. http://www.spec.org/web2005/
18) The Apache Software Foundation: Apache Tomcat. http://tomcat.apache.org/
19) Aladdin Enterprises: RFC1321-based (RSA-free) MD5 library. http://sourceforge.net/projects/libmd5-rfc/
20) Kerdlapanan, D. and Khunkitti, A.: Content-based load balancing with multicast and TCP-handoff, *Proc. 2003 International Symposium on Circuits and Systems, 2003. ISCAS '03.*, Vol.2, pp.II–348–II–351 (2003).
21) Papathanasiou, A. and Van Hensbergen, E.: KNITS: Switch-based Connection Hand-off, *INFOCOM 2002. 21st Annual Joint Conference of the IEEE Computer and Communications Societies. Proc. IEEE*, Vol.1, pp.332–341 (2002).
22) Marwah, M., Mishra, S. and Fetzer, C.: Fault-tolerant and scalable TCP splice and web server architecture, *IEEE Symposium on Reliable Distributed Systems*, pp.301–310 (2006).

(Received July 26, 2010)
(Accepted November 24, 2010)

Hajime Fujita is a Project Research Associate at Information Technology Center of the University of Tokyo. He received his Master of Computer Science degree in 2008 from the University of Tokyo. His research interests include systems software for computer clusters and dependable systems.

Yutaka Ishikawa is a professor of the University of Tokyo, Japan. He received his B.S., M.S., and Ph.D. degrees in electrical engineering from Keio University. From 1987 to 2001, he was a member of AIST (former Electrotechnical Laboratory), METI. From 1993 to 2001, he was the chief of Parallel and Distributed System Software Laboratory at Real World Computing Partnership. His interests include dependable parallel and distributed systems and the next generation supercomputer.