

Distributed TCP: An Efficient Infrastructure to Support Persistent Connections

Ravindranath Kokku[‡] Ramakrishnan Rajamony[†] Lorenzo Alvisi[‡] Harrick Vin[‡]

[‡] Dept. of Computer Sciences [†]Austin Research Lab
U.T. Austin, USA IBM Austin, USA

IBM Mentor: Ramakrishnan Rajamony

Abstract

Systems based on a cluster of off-the-shelf workstations offer a cost-effective hardware platform for designing high-performance network servers. Critical to the performance of these systems is the ability to distribute efficiently client requests among the set of workstations that constitute the back-ends in the cluster. This problem becomes especially challenging when clients and servers communicate through *persistent connections*, which allow multiple requests to be issued along a single TCP connection to the server. This paper presents a new protocol, called *partial hand-off*, that aims at addressing this challenge. Our initial simulations show that partial hand-off scales well, and for persistent connections, can significantly outperform current request distribution protocols. A prototype implementation of partial hand-off in the Linux kernel is under development.

1 Introduction

Server clusters are behind most of the dynamic web services, e.g. e-commerce and web personalizations, deployed on the Internet today. Such systems are attractive because they combine the price-performance characteristics of commodity hardware with the ability to increase server throughput by adding more server nodes to the cluster. A fundamental problem that these systems need to address is scalability. Today, because of the difficulty in balancing the load on the servers within a cluster, the number of client requests that a cluster can service does not grow linearly with number of servers.

Recent research has shown that throughput and overall cluster resource utilization can be improved by distributing client requests to different servers on the basis of the request's content. For instance, a client request to access object O should be forwarded to a server that has a copy of O in its cache, rather than to one that would have to fetch O from disk. Content-based techniques for request distribution are simple to implement and perform quite well when each client request is handled by a separate client-server connection [22]; however, when a single connection is used to handle multiple, possibly unrelated, client requests, (through a so called *persistent connection*) content-based techniques become significantly more complex and expensive [6, 24].

In this paper, we propose a new content-based approach to request distribution that performs well with persistent connections. We call this approach *distributed TCP (DTCP)*. DTCP uses a mechanism that we call *partial hand-off* to shift an already established TCP connection from one server to another in a way that is transparent to the client. This mechanism enables each request to be handled by the server best equipped to do so, and reduces the overhead of handling the request. As a result, in our simulations DTCP consistently outperforms current request distribution protocols for persistent connections: for a cluster of 15 servers, DTCP yields anywhere between 25% and 100% higher throughput, depending on the traces being used. We are currently implementing our design in the network stack (TCP level) of the Linux operating system.

The rest of the paper is organized as follows. Section 2 formulates the problem and describes existing solutions and their limitations. Section 3 presents DTCP and discusses how it compares to existing solutions. To evaluate DTCP, we have performed a series of simulations, whose results are reported in Section 4. Section 5 describes a prototype implementation of DTCP. We cover related work in Section 6 and conclude in Section 7.

2 Problem Formulation

As we noted before, recent research has shown that distributing client requests among the servers in a cluster on the basis of the request’s content can dramatically improve the cluster’s throughput. For instance, in the Locality Aware Request Distribution (LARD) protocol [22], a front-end server sends each request to the back-end server that has the highest probability of having in its cache the data required to respond to the request.

The effectiveness of content-based request distribution protocols, however, depends significantly on the protocol used by clients and server to manage requests. The two protocols most used on the web today are HTTP/1.0 [3] and HTTP/1.1 [4]. Each of these protocols presents unique challenges for achieving a highly scalable cluster architecture.

In HTTP/1.0, each client request generates a separate TCP connection. This makes it easy to use content-based protocols, because, for each request, it allows the client to establish a connection with the most appropriate server. However, the overhead involved in setting up and tearing down a connection for each request hurts scalability [20, 21], and increases the latency of a server’s response.

To reduce this overhead, HTTP/1.1 allows clients to send multiple HTTP requests to a server through a single TCP connection [4]. The server keeps open its connection to a client for a configurable idle time (typically 15 seconds) after the last request received from that client. By making connections persistent, HTTP/1.1 amortizes TCP establishment and tear down costs and increases network utilization by avoiding multiple TCP slow starts. Unfortunately, persistent connections can also limit the effectiveness of content-based request distribution policies, as we describe below.

2.1 Content-based request distribution and persistent connections

Consider the following naive application of content-based request distribution to persistent connections. When a persistent connection is established, content-based criteria are applied to the first request to determine the back-end server that should handle the connection.

Once a back-end server (say, BE1) is selected, it handles every subsequent request received through the connection. This simple scheme works well as long as all requests coming through the persistent connection can be served efficiently by BE1. For instance, these requests may refer to an html page and to all the data objects embedded in that page: it is likely that if one of these objects is best served by BE1, so will be the others. In general, if the requests sent on a persistent connection tend to occur always together (i.e. if they are *related* to each other) it is possible to keep the data they request in the cache of the same back-end server, thereby ensuring good response time.

The problems arise when the requests are not related. Suppose, for instance, that the client initiating a persistent connection is a proxy cache, as in the scenario illustrated in Figure 1. Let C1 and C2 be two clients that send requests simultaneously through a proxy P to a server whose front-end is FE and two of whose back-ends are BE1 and BE2. Suppose that C1 and C2 send requests for objects A,B,C,D, and E. BE1 has objects A,B, and C in its cache, while BE2 has D and E. The connection between proxy P and BE1 is established based on the request A, as A is the first request received. Since a request for D is received on the same connection, it is server BE1, and not BE2, that receives the request for D, even though it is BE2 that can serve the request more efficiently.

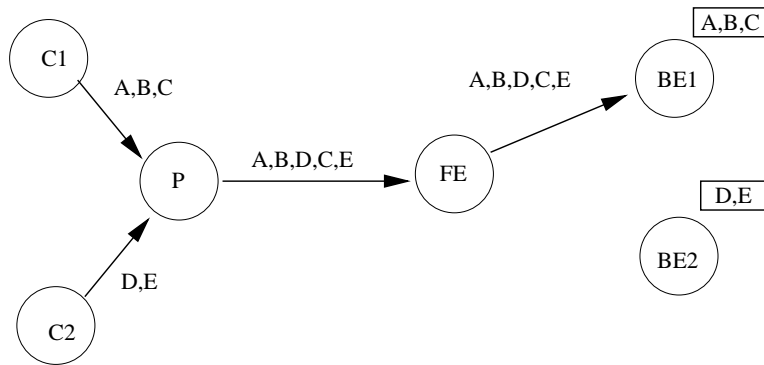


Figure 1: Effect of persistent connection on locality

To quantify the degree to which proxies may contribute to worsen the problem with persistent connections, we analyzed a few client proxy traces available to us. Using a particular client proxy trace taken over 24 hours, we extracted entries for servers that received the highest number of connections from this proxy. For each such server, we extracted the information shown in Figures 2 and 3. A careful analysis of the proxy traces shows that, since proxies group requests from different clients on the same connections and these clients often ask for different data, there is a very good chance that requests from different clients on the persistent connection be not related.

Table 1 shows the results of the analysis done on IBM client proxy traces. *Total Reqs.* is the number of requests that the server received from this proxy on behalf of the clients behind the proxy. *Unrelated Reqs.* is instead the number of requests that are received from a client that didn't open the connection with the server (i.e. requests sent on a connection that was opened by proxy because of a request from some other client). The value *% Unrel.* is derived from the number of unrelated requests and the total requests. *Pers. Conn.* and *Clients* give, respectively, the number of persistent connections and clients, and are included

Server	Total Reqs.	Unrelated Reqs.	% Unrel.	Pers. Conn.	Clients
1	23407	5310	22.68%	519	281
2	10748	2844	26.46%	118	50
3	4168	548	13.15%	1451	78
4	3533	996	28.19%	1073	50
5	2313	421	18.20%	837	50

Table 1: IBM proxy trace analysis

Server	Total Reqs.	Unrelated Reqs.	% Unrel.
1	6136	1425	23.22%
2	4421	1311	29.65%
3	3629	1737	47.86%
4	3373	984	29.17%
5	3182	1192	37.4%
6	3077	1151	37.41%

Table 2: Squid proxy trace analysis

to give an idea of how the total number of requests were sent to the servers.

Table 2 shows the same results for Squid proxy traces [2]. Both tables show that there is a significant number of unrelated requests.

2.2 Existing solutions and limitations

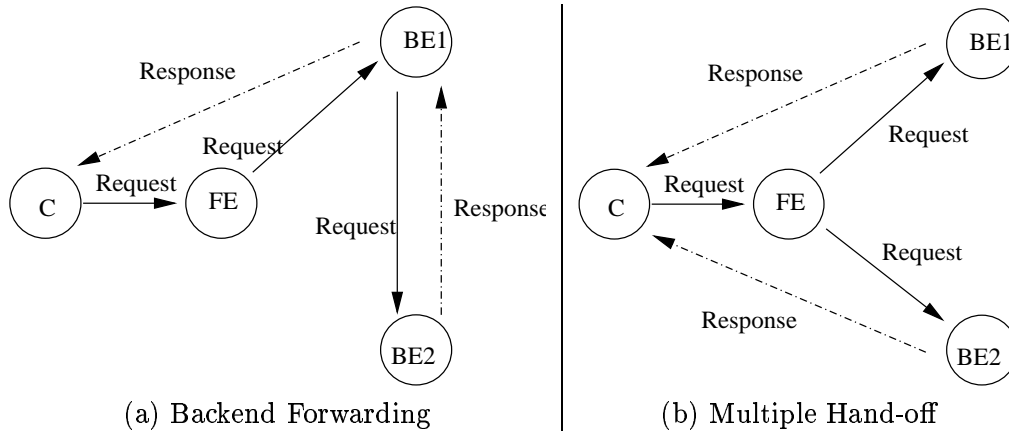


Figure 2: Existing mechanisms for request distribution.

Aron et al.[6] suggest two mechanisms to address the problems generated by persistent connections: (1) back-end forwarding and (2) multiple hand-off (Figure 3). In back-end forwarding, when a request arrives at BE1 on a persistent connection with the extra information that this request would best be served by BE2, BE1 requests the content directly from BE2. Then, on receiving the response from BE2, BE1 forwards it to the client. The disadvantage of this approach is the back-end network may become a potential bottleneck. Though backend forwarding performs fairly well in for today's workloads, our study with

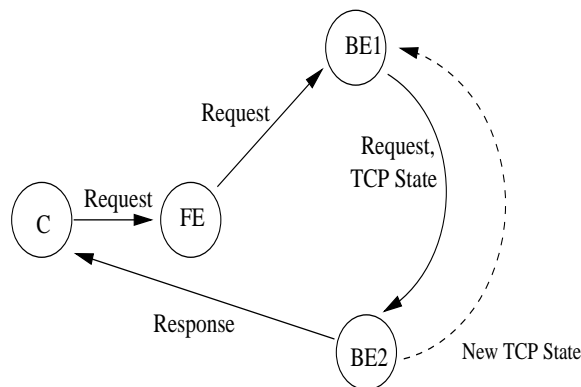


Figure 3: DTCP request distribution mechanism

SpecWEB99 [10] traces (see Section 4) suggests that future workloads will need better strategies for request distribution.

In TCP hand-off, the server-side connection is handed-off to a different back-end node. Once the state is transferred, the new back-end directly transmits responses to the client. The original back-end is still responsible for forwarding TCP acknowledgement packets (TCP acks) to the new back-end. Though TCP-ack forwarding seems to exact only a small overhead, if the connection gets handed-off multiple times, a chain of servers may become unnecessarily loaded (for ack forwarding). This problem can be overcome but with a more complex protocol. TCP hand-off runs as a loadable kernel module over the TCP stack and not at the TCP level. Figure 6 shows how multiple handoffs can create a long chain of back-end nodes getting loaded because of an already handed-off connection.

3 Our approach

3.1 Key Idea

We assume a typical web server architecture wherein there is a front-end server (a layer-4 switch or an intelligent dispatcher) visible to the client and a few back-end servers invisible to the clients. The key idea of our approach is a *partial hand-off* of the connection. In this approach, the TCP state of a connection to a back-end server is transferred to a new back-end if the latter can best service the request. The TCP connection state includes the client IP address and port number and TCP window information (such as which packet to be sent, last packet received, last ack sent, last ack received etc.). This information is required to continue sending packets on a connection. The (new) state is transferred back to the original back-end once the new back-end is done sending the response. Since the connection is not handed-off completely, but is passed back and forth when needed, we call the approach *partial handoff* (Figure 4).

Figure 5 shows qualitatively how a scheme like DTCP can reduce the amount of data that traverses the TCP stack. In back-end forwarding, the response data unnecessarily traverses BE1's TCP stack twice, while in DTCP this does not happen.

The main disadvantage of our mechanism is that it needs an OS specific implementation

since DTCP resides as a part of network stack at the TCP level.

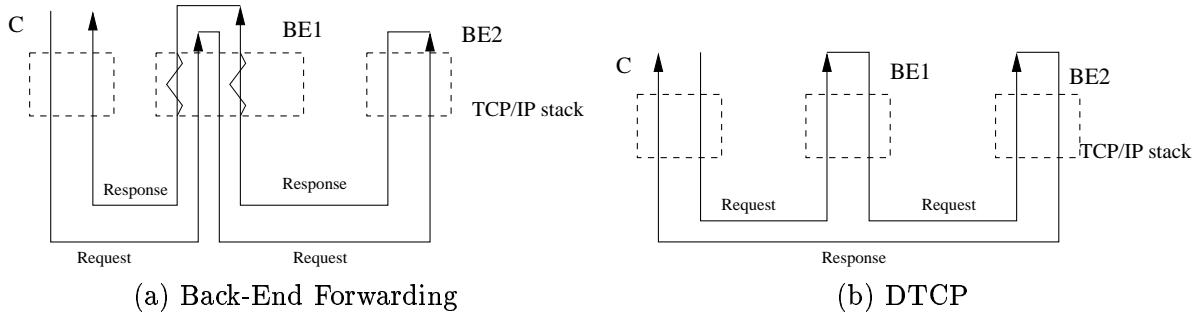


Figure 4: Effect on Network stack.

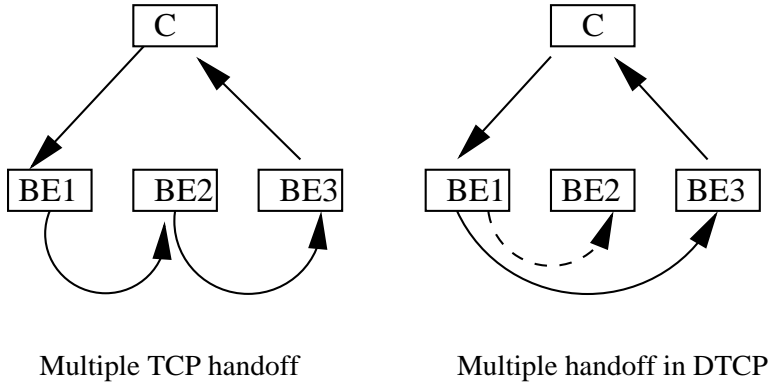


Figure 5: Comparison of LARD's TCP handoff and DTCP

4 Simulation

Before implementing a prototype of DTCP, we built a simulator to get an first sense of how our approach compares with the current state-of-the art. We performed this comparison using both real workloads and the synthetic SpecWEB99 traces. The next section describes the simulator implementation and the results we have obtained with it.

4.1 Implementation

Our simulator is built in CSIM, a process oriented discrete event simulation package used for C programs [18].

Each back-end node is modeled to contain a cpu, a disk, two input adaptors and two output adaptors(Figure 7). All the above devices are modeled in CSIM as independent processes. This helps us in simulating the effect of the devices functioning in parallel on different requests. For example, a cpu may be serving a request when a disk is busy fetching a file for another request at the same time. Communication between any two devices is through *mailboxes* [18]. Each device owns a mailbox and only the owner can read its mailbox; other devices can only write to the mailbox. If device D_a wants to send a

message to device D_b , D_a writes into the mailbox of D_b . Each node has a cache of most recently accessed files that is maintained as a linked list stored in LRU order.

4.2 Architecture

Our simulator is based on an architecture of cluster-based web servers which contains a (set of) front-end(s) and a set of back-ends. The front-end on receiving client requests, distributes them to one of a set of back-ends according to either DTCP's or LARD's request distribution policies. We chose to compare DTCP with LARD with back-end forwarding because the latter is the best solution to the persistent connection problem today: other policies like round robin and weighted round robin have already been shown to be less efficient than back-end forwarding.

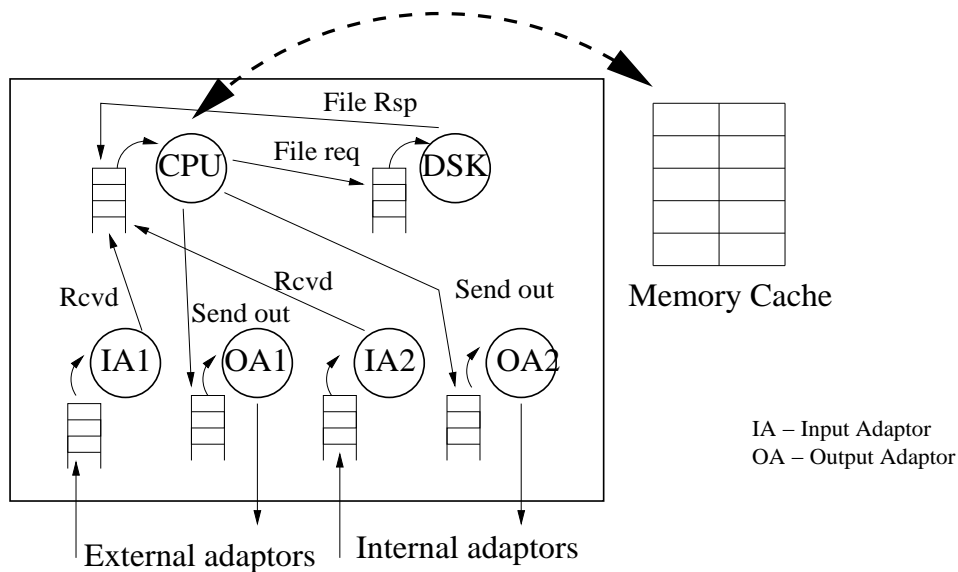


Figure 6: Simulator architecture

The simulator simulates optimal behavior of LARD and DTCP policies. In order to send a request to a specific back-end, the front-end looks into the data structure that represents the cache of the back-end and forwards it to a node only if it is sure that the request will get serviced from the cache. If no node exists which caches the file in the memory or if the nodes that cache the file in their caches are overloaded with requests, then the least loaded node is selected to serve the request, based on the number of disk requests queued at the node. The exact behavior of the load balancing algorithm and choice of nodes for serving requests is described below.

4.3 Functionality

The front-end maintains a server set for each file requested - each file maps to a set of servers whose caches contain the file. The load on the nodes is defined in terms of number of messages lying in the mail boxes at the instant we make our decision. In the simulator,

we define two types of load for a back-end node- *Disk load* and *Processing Load*. Disk Load is the number of messages in the disk's mailbox. Processing Load (outstanding) is the total number of messages in cpu and the two input adaptors of a node. The messages in the output adaptors need not be counted because we can assume that they will not cause any further load on the CPU of that node. Algorithm 1 describes the node selection algorithm at the front-end.

Algorithm 1 Node Selection Algorithm

```

1: while Server Set is not NULL do
2:   M = node with least Processing Load
3:   if File not in M's cache then
4:     Server Set = Server Set -{M}
5:   else
6:     break
7:   end if
8: end while
9: N = node with least Disk Load
10: if req. of M - req. of N > 100 && N not in Server Set then
11:   Server Set = Server Set + {N}
12:   Preferred nodeId = N
13: else
14:   Preferred nodeId = M
15: end if

```

To simulate persistent connections, if two requests arrive from the same client within 15 seconds of real time (as recorded in the server trace timestamps), we assume that the requests have arrived on the same persistent connection.

The front-end keeps track of persistent connections and forwards all requests on the persistent connection to the same back-end together with the id of preferred back end, i.e. the back-end that would be the most efficient in servicing the request. The decision is left to the first back-end whether to get the service from the preferred back-end. Each back-end also keeps track of the persistent connections so that connection establishment and teardown costs are correctly accounted for.

4.4 Simulation details

The costs for the basic request processing steps were taken from the LARD paper [22]. According to their measurements, on a 300MHz Pentium II machine, running FreeBSD and an experimental web server, connection establishment and teardown costs are set at $145\mu\text{s}$ and transmit processing incurs $40\mu\text{s}$ per 512 bytes. Each disk request takes 14ms seek time and offers 10MBps data transfer rate. The amount of main memory on each node is kept as 150MB. (a 256 MB RAM, after accounting for space taken by OS and web server and other appls).

In Back-End Forwarding, if the request is forwarded to the preferred back-end, the preferred back-end sends back the response to the server that sent the request. The original back-end caches the file that it received from the preferred back-end and serves the request.

If the file size is more than 500KB (as used by LARD), the file is not cached. In DTCP, if the request is forwarded to the preferred back-end, the preferred back-end directly serves the request. It follows that, for a back-end forwarded request in LARD, the original node charges twice for the processing of the forwarded request and its response. This is required to simulate the effect in the real scenario in which data travels up and down the TCP/IP stack unnecessarily at the original back-end.

4.5 Results

In order to evaluate the performance of DTCP, we used a range of real traces which are representative of different types of servers. We measure the performance of our strategy in terms of cluster throughput (number of requests serviced per second). The attached graphs show the throughput results obtained when the simulator was run on the following traces[1].

1. CLARKNET - Traces of ClarkNet WWW server. ClarkNet is a full Internet access provider for the Metro Baltimore-Washington DC area.
2. NASA - Traces at NASA Kennedy Space Center WWW server in Florida.
3. Olympics 98 - Traces of server cluster used to host Olympics 98 web site.
4. UTCS web server - Traces at University of Texas Computer science department.
5. World cup 98 - Traces of server cluster used to host World cup soccer 98 web site.
6. SpecWEB99 - Web server trace where the clients generate a SpecWEB99 workload.

All the graphs in Figure 8 show that there is a consistent performance improvement by implementing a technique like DTCP. The increase in performance as the number of server nodes increases shows that DTCP is scalable too. The first five graphs shows that even today's workloads can benefit from a DTCP infrastructure. SpecWEB99[10] traces model extreme experimental conditions because of the random request correlation.

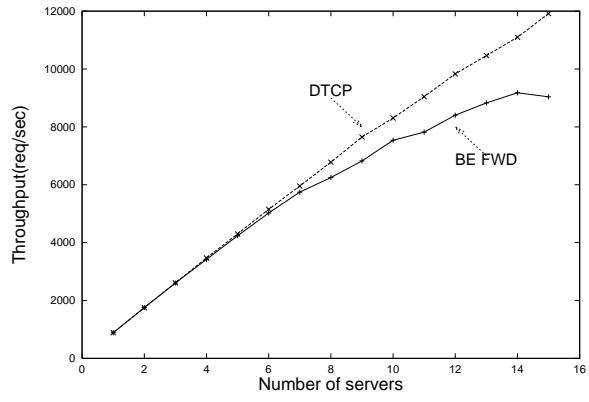
The graph in Figure 9 shows average data transferred on the back-end network with the increase in size of the cluster. The amount of back-end data in most workloads is very less compared to that of SpecWEB99. The graph shows that for web requests that would look more like SpecWEB99, we would need better schemes than back-end forwarding in order to get better scalability.

5 Prototype DTCP

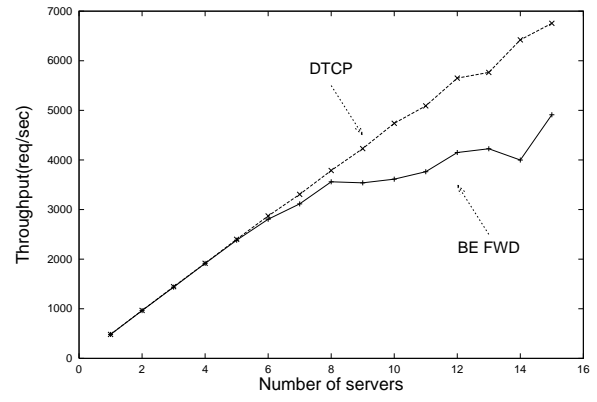
5.1 Implementation

DTCP is a modification to the network stack at the TCP level. It is based on the *partial handoff* approach wherein we handoff the connection to the server that should serve the request and then get back control once its done. We do this by transferring the connection state(TCP state data) from one node to another wherever and whenever required.

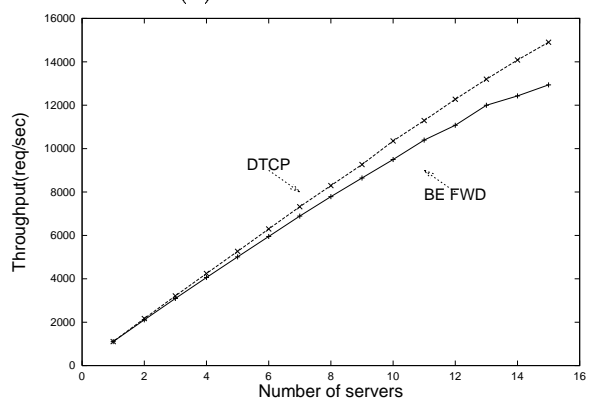
We introduce the concept of an active and a passive TCP layer. The *passive layer* is a stripped down version of the traditional TCP layer that is spawned on the remote node that



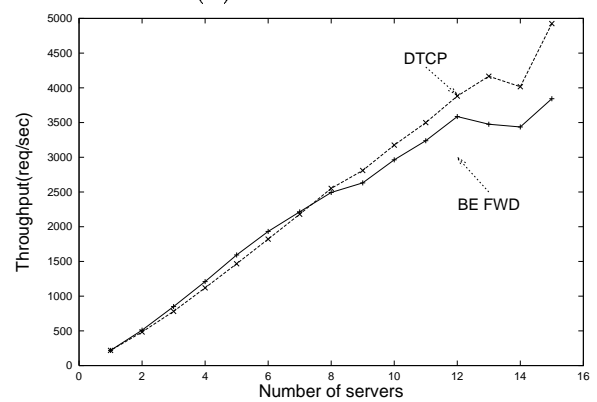
(a) Clarknet server



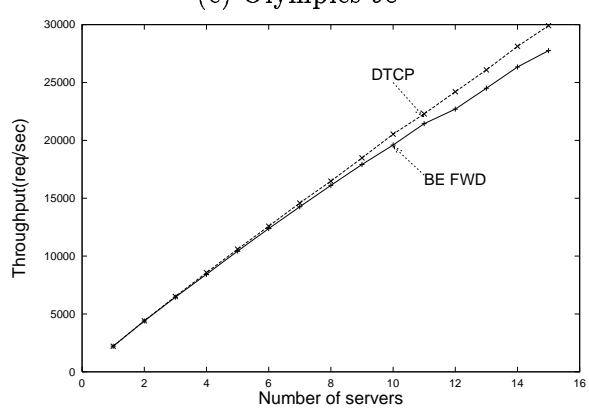
(b) NASA server



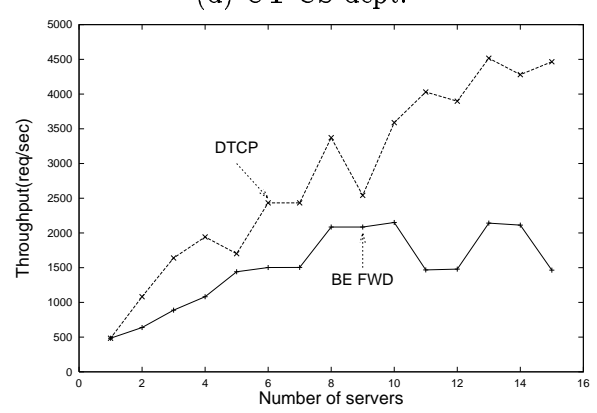
(c) Olympics 98



(d) UT CS dept.



(e) World cup 98



(f) SpecWEB99

Figure 7: Throughput comparison of back-end forwarding and DTCP.

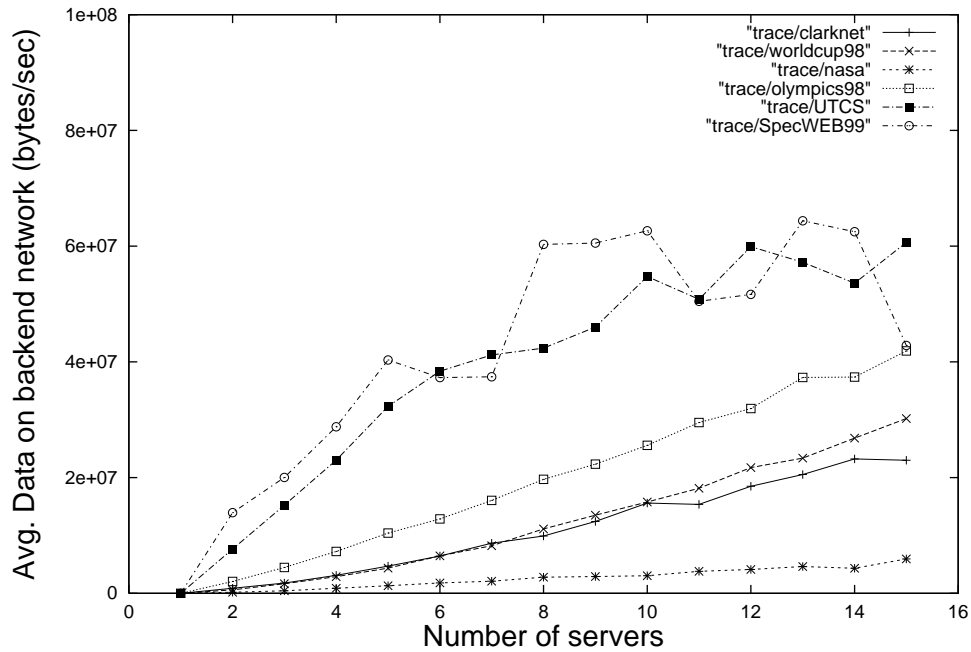


Figure 8: Average data transferred on back-end network

can best service the request. The active layer is the one that has a direct TCP connection setup with the client.

The passive layer doesn't participate in three-way handshake for connection setup. The passive layer is activated on the remote node on request from the active layer and is responsible for sending the requested data to the specified client and maintaining its version of the TCP window.

The active layer centrally maintains the state of the connection, and hence it maintains the window information too. When the connection is handed-off, the window information is sufficient enough to be able to send acknowledgement and retransmission requests.

The passive layer, on receiving the response from the application immediately informs the active layer how many packets it would take to transmit the response. The active layer updates its window information so that acks or retransmissions for those packets are sent to the passive layer accordingly.

The acknowledgements are received by the active layer and forwarded to the passive layer. On receiving the acks for data packets, the passive layer sends out the next packet.

The timers for the whole connection are maintained on the active layer. If a retransmission is required, the active layer informs the passive layer to retransmit a packet.

The communication between active and passive layer is over IP. We include special TCP options to recognise these special communication packets.

The implementation provides an API to the web server application through a *hand-off(socket, request, passive node)* system call. Passive node here is the node that can best serve the request. When an application calls the handoff system call, a new passive socket is opened on the passive node and the TCP state is transferred to the new passive socket.

The application on the passive side is informed of arrival of a request. When the application returns a response, the response is sent by the passive node using the passive socket information. Once all the packets of the response are sent, the passive side closes the socket and informs the active socket about the completion of the service.

Subsequent requests from the application on the active side are queued till the passive side returns after closing the socket. The active side after receiving a passive close indication from the passive side, then services the subsequent requests if queued or waits for the application to send another request.

We are currently implementing the active and passive layers in a LINUX network stack.

6 Related Work

Aron et al.[6] have devised mechanisms for efficiently supporting persistent connections in cluster based systems. They proposed two mechanisms - back-end forwarding and TCP multiple handoff. In back-end forwarding, the data that is needed by a persistent connection and is available in another node's cache, is obtained from the end node's cache instead of fetching from the disk. Our mechanism instead of fetching the data and then serving it, hands-off the connection for some time and asks the back-end node that has the file to service the request. Our mechanism tries to reduce the amount of data traversed in a back-end's TCP stack because of back-end forwarding.

TCP multiple handoff[6] may lead to a chain of nodes getting involved in serving a client even after the connection is handed-off, thus reducing the performance of the server cluster. Our mechanism avoids this chain by only partially handing off the connection.

In [23] the authors try to solve the persistent connection problem by bundling objects together. Their idea is complementary to ours and can be used in conjunction with ours to gain improved performance.

In [14], the author describes several performance problems resulting from interactions between implementations of persistent-HTTP (P-HTTP) and TCP. They describe each problem and potential solutions. The solutions are complementary to our work and can be used over our DTCP infrastructure.

Hunt et al. [15] presented a mechanism to handoff active connections which is similar to the TCP single handoff proposed by Pai et al.[22]. The implementation of such a mechanism is not known to be done. Our implementation of DTCP uses the TCP options to communicate with the passive stack. But our protocol is more sophisticated to support multiple partial handoffs.

Microsoft's Network load balancing [9] transparently partitions the client requests among the hosts and let the clients access the cluster using one or more virtual IP addresses. The cluster looks as a single server to the client. Extra servers can be transparently added to the cluster. But their strategy is not scalable to a large number of nodes, as they depend on broadcast of all packets to all nodes and expect that packets not meant for them will be discarded by the cluster nodes.

Network Dispatcher(ND) [16] is a TCP connection router that supports load sharing across several TCP servers. It provides a fast IP packet-forwarding kernel-extension to the TCP/IP stack. However it was not designed to effectively handle persistent connections.

Many cluster based servers and routers are widely deployed on the Internet[8, 17, 19]. These servers implement strategies like round robin, weighted round robin[12], source based forwarding[11] and hardware translation of network addresses [17]. But none of them seem to implement an efficient solution for persistent connections.

7 Conclusions

In this paper, we described a problem in cluster based web servers that occurs because of increased data transfer on persistent connections and because of increased number of proxies on the Internet. We presented a solution based on partial hand-off to tackle such problems. A simulation study showed that our strategy is highly scalable and can provide increased server cluster performance as compared to the existing best strategies. For example, we could achieve 25% better throughput with 15 servers when tested with various web traces representative of today's wide range of workloads. Also, with SpecWEB99 workloads, which are representative of future Internet traffic, we found that we could achieve upto 100% improvement in performance. The results are yet to be supported by a prototype implementation. We are currently implementing the distributed TCP mechanism in the Linux kernel.

References

- [1] Internet archive of http server logs. <http://ita.ee.lbl.gov/html/traces.html>.
- [2] Squid proxy traces. <ftp://ftp.ircache.net/Traces/>.
- [3] RFC 1945. Hypertext transfer protocol – http/1.0.
- [4] RFC 2068. Hypertext transfer protocol – http/1.1.
- [5] Akamai. <http://www.akamai.com>, 2000.
- [6] M. Aron, P. Druschel, and W. Zwaenepoel. Efficient support for p-http in cluster-based web servers. *In Proceedings of the Usenix 1999 Annual Technical Conference, Monterey, CA.*, June 1999.
- [7] Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore W. Hong. Freenet: A distributed anonymous information storage and retrieval system. July 2000.
- [8] IBM Corporation. Interactive network dispatcher. <http://www.ibm.com>.
- [9] Microsoft Corporation. Network load balancing technical overview. <http://www.microsoft.com>., 2000.
- [10] Standard Performance Evaluation Corporation. Specweb99 benchmark. <http://www.spec.org>.
- [11] O. Damani, P. Chung, Y. Huang, C. Kintala, and Y. Wang. Oneip: Techniques for hosting a service on a cluster of machines. *Computer Networks and ISDN Systems, 29:1019-1027.*, 1997.

- [12] A. Fox. Extensible cluster-based scalable network services. *Proc. 16th ACM Symposium on Operating Systems Principles (SOSP-16)*, St. Malo, France., Oct 1997.
- [13] Gnutella. <http://gnutella.wego.com>, 2000.
- [14] John Heidemann. Performance interactions between P-HTTP and TCP implementations. *ACM Computer Communication Review*, 27(2):65–73, April 1997.
- [15] G. Hunt, E. Nahum, and J. Tracey. Enabling content-based load distribution for scalable services. Technical report, May 1997.
- [16] Guerney D.H. Hunt, German S. Goldszmidt, Richard P. King, and Rajat Mukherjee. Network dispatcher: a connection router for scalable internet services.
- [17] Cisco Systems Inc. Localdirector. <http://www.cisco.com>.
- [18] Mesquite Software Inc. Csim18. <http://www.mesquite.com>.
- [19] Resonate Inc. Resonate dispatch. <http://www.resonateinc.com>.
- [20] J. Mogul. The case for persistent-connection http. *In the Proceedings of the ACM SIGCOMM '95 Symposium*, 1995.
- [21] V. Padmanabhan and J. Mogul. Improving http latency. *In the Proceedings of the Second International WWW Conference, Chicago, IL*, Oct 1994.
- [22] V. 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 ACM Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, CA.*, Oct. 1998.
- [23] Craig E. Wills, Mikhail Mikhailov, and Hao Shang. N for the price of 1: Bundling web objects for more efficient content delivery. Technical report, Nov 2000.
- [24] X. Zhang, M. Barrientos, J. 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.