Splitting HTTP Requests on Two Servers

Bharat S. Rawal,
Department of Computer and Information Sciences
Towson University
Towson, Maryland, USA
brawal@towson.edu

Ramesh K. Karne
Department of Computer and Information Sciences
Towson University
Towson, Maryland, USA
rkarne@towson.edu

Alexander L. Wijesinha
Department of Computer and Information Sciences
Towson University
Towson, Maryland, USA
awijesinha@towson.edu

***Abstract*—Many techniques are commonly used to increase server availability or for distributing the load among a group of servers. We propose a technique for splitting a single HTTP request that allows a TCP connection to be dynamically split between two Web servers without using a central control. For example, one server can handle connection establishment and closing, while another handles the data transfer. This approach requires no client involvement since the existing connection with the initial server continues to be maintained, and the client is completely unaware of the splitting. We demonstrate the splitting concept in a LAN environment and provide related performance results that highlight several interesting features of splitting. The splitting was done using two bare PC servers with no operating system (OS) or kernel running in the machines. Splitting also works with clients located anywhere on the Internet, although servers have to be located on the same LAN. Our implementation and results indicate the feasibility of splitting TCP connections to transparently redistribute server load without client involvement.**

Keywords – Split TCP Connections, Bare Machine Computing, Application-Object, M-TCP, Web Server, and Performance.

I. INTRODUCTION

Web server reliability and load distribution among Web servers are important problems that continue to be addressed using a variety of techniques. In particular, load balancing techniques are used at various layers of the protocol stack to share the load among a group of Web servers [19], [20]. Alternatively, in approaches such as Migratory TCP (M-TCP) [13], a client connection is migrated to a new server and an adaptation of TCP enables migration of the connection state. We propose a technique for splitting a TCP connection between Web servers that allows one server to handle connection establishment and another to handle data transfer. Splitting also allows a server to self-delegate a percentage of its connection requests to another server for data transfer. Splitting TCP connections in this manner enables servers to share the load without a central control and without any client involvement.

To explain splitting, we note that in essence, Web servers process client requests and provide data (resource) files to clients. These files are transferred via HTTP that runs as an application-level protocol on top of TCP. A typical message exchange over TCP to handle a single HTTP request (such as a GET request) is shown in Figure 1. The HTTP and TCP protocols are shown in an intertwined manner corresponding to the coupling of these protocols in a bare machine (or bare PC) Web server [5] that runs applications without an operating system (OS) or kernel. Protocol intertwining in bare machine computing (BMC) has been previously shown to improve Web server performance [5], [6]. The application-centric BMC Web server design and architecture [5] includes protocols as part of the application and facilitates inter-layer protocol communication. In this paper, we exploit the novel BMC Web server design to split a single TCP connection among servers in a manner that is totally transparent to the client, except possibly for a small increase in latency.

To explain how a TCP connection is split by a BMC Web server, consider the message exchange in Figure 1. When an HTTP GET request arrives after the TCP connection is established, the BMC Web server sends a response (GET-ACK) to the client, and updates the client's state. The GET request is then processed by the server and the requested file is sent to the client during the data transfer. This process differs somewhat based on whether the HTTP request is static or dynamic. Although this paper addresses splitting only for static requests, the techniques apply to dynamic requests as well. Once the data transfer is complete, the connection is closed by exchanging the usual FIN and FIN-ACK messages. Although multiple GET requests can be sent using a single TCP connection, for ease of explanation, we only consider the case of a single GET per connection. A single HTTP request and its underlying TCP messages can thus be divided into connection establishment (CE), data transfer (DT), and connection termination (CT) phases as shown in Figure 1.

The novel splitting concept presented in this paper is based on splitting the HTTP request and underlying TCP connection into two sets {CE, CT} and {DT}. This split allows one server to handle connections and the other to handle data without a need for too many interactions between servers. The data could reside on only one server or on both servers if reliability is desired. Splitting a client's HTTP request and the underlying TCP connection

978-1-4244-8953-4/11/$26.00 ©2011 IEEE

in this manner also provides the additional benefit of data location anonymity in addition to enabling load sharing among servers. Furthermore, server machines optimized to handle only connection requests and others optimized to handle only data transfer could be built in future to take advantage of splitting.

The experiments described in this paper to demonstrate split connections were done with servers and clients deployed on the same LAN. However, splitting can be also used (without any modification to the servers) when clients are in a WAN or Internet environment provided the servers are on the same LAN. We have conducted experiments to verify that splitting works with clients on different networks communicating with servers through a series of routers. The reasons for requiring servers to be on the same LAN are discussed further in Section III.

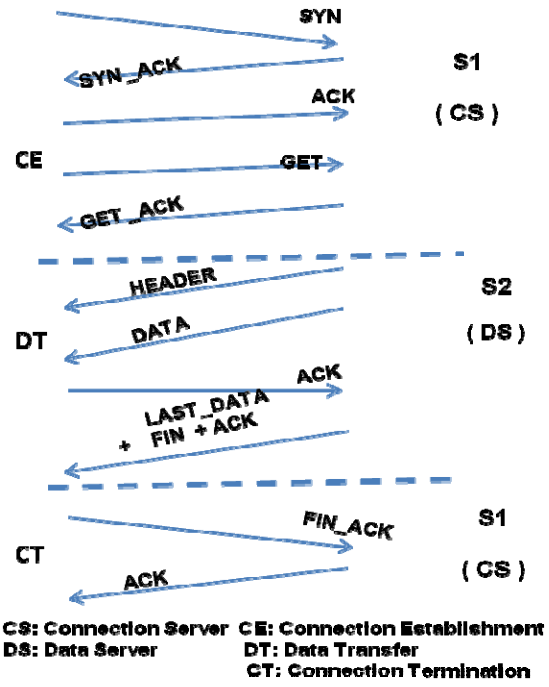

Figure 1. HTTP/TCP Protocol Interactions

The rest of the paper is organized as follows. Section II presents related work, and Section III describes the split architecture. Section IV describes our design and implementation, and Section V presents the measurements conducted to evaluate performance of split servers. Section VI contains the conclusion.

II. RELATED WORK

Bare Machine Computing (BMC), previously referred to as dispersed OS computing [7], enables an application to run on a bare PC without any centralized kernel or OS support. In the BMC approach, an application encapsulated in an application object (AO) [9] directly communicates to the hardware. BMC is similar to approaches such as Exokernel [3] (which moves kernel code to an external kernel), IO-Lite [12] (which devises mechanisms to by-pass system calls), Palacio [10] (which creates a lightweight kernel for super-computing applications), Libra [1] (which uses an efficient kernel for Java based applications), factored OS [15] (which proposes next generation OS structures to run a large number of multi-core processors), bare-metal Linux [14] (which provides by-pass mechanisms in Linux for the boot process), and TinyOS [18] (which is a lean kernel for embedded systems).

Reducing OS or kernel overhead has an immediate impact on performance. For example, a high performance gain may be achieved by moving code from kernel to an application level [4]. The difference between these approaches and the BMC approach is that the latter does not use a kernel or OS to control applications. BMC treats the application as a single entity all the way from the application to the hardware level and includes direct hardware interfaces within the application itself [8]. The thread of execution in BMC is statically designed by the AO programmer. That is, each single thread of execution in the AO runs until its end and returns to the main task by suspending itself. In BMC, the main task runs whenever no other task is running. A task that is waiting for an event will be resumed when the event or a timeout occurs.

Splitting TCP connections as described in this paper is easy to implement on a bare PC. To the best of our knowledge, a technique for splitting a HTTP-based TCP connection in this manner has not been proposed previously. Splitting is similar to masking failures in TCP [16] and using the M-TCP (Migratory TCP) protocol to migrate TCP connections from one server to another [13]. However, connection migration in M-TCP does not involve splitting an HTTP request and TCP connection, or operating in split mode. Moreover, the client needs to initiate the migration process in M-TCP, whereas TCP connection splitting is transparent to the client. TCP connection splitting is also different from migrating Web applications in mobile computing [2], which moves applications from one node to another on the Web by using virtualization techniques. Likewise, connection splitting is different from process migration in [11], which requires that an agent be dispatched from one system to run on another at a remote location.

TCP connection splitting and the different techniques used for load balancing share some common characteristics. For example, both load balancing and TCP connection splitting enable server load to be distributed among servers. In layer-2 (link layer) load balancing, multiple links act as a single virtual link, and in layer-4 (transport layer) load balancing, requests are redirected based on port numbers and IP addresses. In layer-7 load balancing [19], content information from the HTTP request is used to make load balancing decisions. In this case, there are two TCP connections that are spliced at the IP layer to reduce overhead: the first between the client and the proxy that receives the initial request, and the second between the proxy and the server that will service the request.

Load balancing could also be based on NAT or DNS information, or deployed in a LAN environment by having a central load balancer select a server and directly forward requests (by using the server's MAC address) as described in [20]. An advantage of TCP connection splitting over such load balancing techniques is that a special switch, proxy or central load balancer is not required since the decision to delegate an HTTP request is made by the server receiving the request. This decision can be made by the server based on its knowledge of its current load and available resources.

III. SPLIT ARCHITECTURE

The split architecture used for the experiments described in this paper is illustrated in Figure 2. Although these experiments were conducted in a LAN environment, as noted earlier, the proposed splitting technique does not require that the set of clients {C} be connected to a LAN (they can be located anywhere on the Internet). The only requirement is that the servers be connected to the same LAN for the reasons discussed below. However, this requirement does not limit the scope or scalability of splitting since many real-world Web server clusters are located within the same LAN. The clients send requests to servers S1 or S2. S1 and S2 are referred to as split servers. For a given request, the connection server (CS) handles the {CE, CT} phases of a connection, and its delegated server S2 (DS) handles the {DT} phase. Similarly, S2 can act as a server for a client's request and its DS will be S1. The clients do not have any knowledge of a DS. A given request can also be processed by the CS without using the DS. In general, there can be a set of n (≥ 2) servers that can delegate requests to each other.

A given request is split at the GET command as shown in Figure 1. The CS handles the connections, and the DS handles the data transfer. In addition to connections, the CS also handles the data ACKs and the connection closing. The CS has complete knowledge of the requested file, its name, size, and other attributes, but it may or may not have the file itself. However, the DS has the file and serves the data to the client. When a TCP connection is split in this manner, the TCP sliding window information is updated by S1 based on received ACKs even though the data file is sent by S2. Likewise, S2 knows what data has been sent, but it lacks knowledge of what data has been actually received by the client. Thus, retransmissions require that ACK information be forwarded by S1 to S2 using delegate messages as described below. The number of delegate messages exchanged should be kept to a minimum since they add overhead to the system and degrade performance.

When a client makes a request to S1, its connection is based on (IP3, SourcePort) (IP1, 80). S1 can serve this request to a client directly, or it can utilize its DS, which is S2, to serve data. The decision to use a DS can be made based on several factors such as the maximum number of requests that can be processed at S1, the maximum CPU utilization at S1, or resource file location.

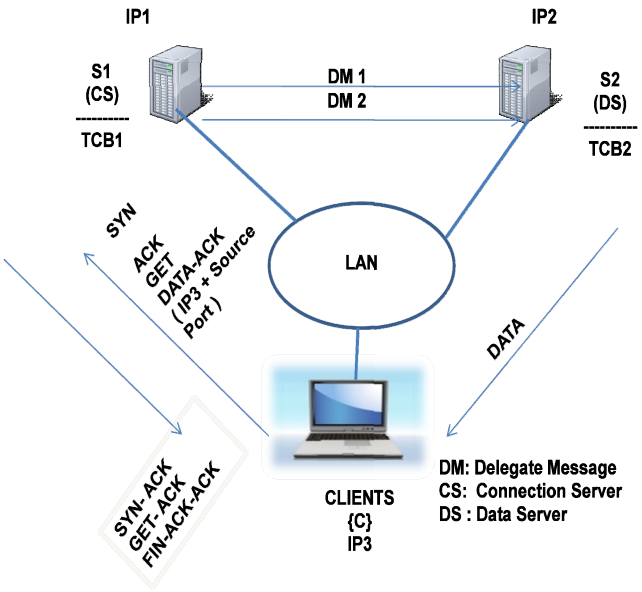

Figure 2. Split Architecture

Alternatively, a load balancing algorithm could be used. When S1 chooses to use S2 as a DS, it proceeds as follows. After the GET command is received, it sends an ACK to the client and also sends a delegate message DM1 to S2 (e.g. DM1). The message DM1 contains the state of the request that is stored in S1 in the form of an entry in the TCP table (referred to as a TCB entry). When DM1 reaches the DS, it creates its own TCB entry and starts processing this request as if it was initiated in the DS itself. When a DS sends data to the client it uses the CS's IP (IP1).

In principle, the Internet setting is not different from a LAN environment since the DS does not need to receive any packets sent by the client to IP address IP1. A client located anywhere on the Internet can communicate in the usual manner with the CS. Since it is unaware that the DS is actually sending the data, it sends the ACKs as usual to the CS with whom the TCP connection was established. From the client's point of view, it has established a connection with IP address IP1. Now consider the information that is present in the local routers and switches assuming that both S1 and S2 are located in the same LAN. Note that only S1 should respond to ARP requests for IP address IP1. This ensures that any local router receiving the response will have a single ARP entry (IP1, MAC S1) in its ARP cache and correctly forward ACKs and connection requests sent with destination address IP address IP1 to S1. Note also that the switch to which S1 is connected will have the entry (MAC S1, INT1) in its forwarding table, where INT1 is S1's interface to the switch. Likewise, the switch to which S2 is connected has the entry (MAC S2, INT2) in its forwarding table, where INT2 is S2's interface to the switch. When S1 sends a delegate message to S2, if they are both on the same LAN, S1 can simply encapsulate the message in a MAC layer frame addressed to MAC S2 (i.e., S2 does not need an IP address to receive delegate messages from S1). Thus, with these assumptions, switches

and routers do not need any special configuration for split connections to work.

However, if S1 and S2 are on LANs with different subnet prefixes (or in general, on WANs or different networks) and communicate through routers, S2 is not reachable using IP address IP1 since its prefix is not consistent with the network it is on. So it will need to use its own IP address IP2 to receive packets including delegate messages from S1. This means that the router for S2 must have an ARP entry (IP2, MAC S2) for forwarding to S2, which will only be present if S2 has responded to ARP request for IP address IP2 with its MAC S2. But in this case, if S2 is also sending data to a client using IP address IP1 as source, it raises a security issue on S2's network due to IP address spoofing. Such spoofing may cause problems with firewalls due to sending topologically incorrect packets. For splitting to work, S2's network must allow S2 to send with IP address S1 and receive with IP address S2; it may also need to send other packets with its own IP address IP2 (S1 sends and receives as usual with IP address IP1). Now if S1 and S2 both delegate to each other, IP spoofing has to be allowed for S1 (on S1's network) as well. There are also TCP issues with splitting due to its sliding window, duplicate acks, fast retransmit, and congestion control that need further study. More delegate messages could be used to address some of these TCP issues, but this would have a negative performance impact.

As the connection and data transfer are split in the architecture, there is a need to send one or more DM messages (DM2s) to DS. At least one DM2 message is needed to indicate that CS received the FIN-ACK. If a received ACK indicates that data is lost, retransmission is needed. One or more DM2s are needed to handle retransmissions since the DS does not receive any data ACKs. The CS monitors the data ACKs and makes a decision to send DM2s as needed. Throughout the splitting process, the client is not aware of DS, and there is no need to involve the client (unlike M-TCP). The last DM2 message to DS is used to terminate when all data has been acknowledged by the client.

Splitting results in two possible overheads. Network traffic due to sending DMs to DS, and the latency encountered at the client due to DM transmission on the LAN (or WAN) from CS to DS. In a LAN environment, this latency is negligible, but may be larger in a WAN or Internet environment. The network traffic generated for each request is at least two DM packets; in most cases it is two packets assuming no retransmissions. If the DM packet is small (168 bytes in a bare PC), the network overhead will be reduced. However, one needs to consider the above two overheads of the split request architecture for a given application.

IV. DESIGN AND IMPLEMENTATION

The detailed design and implementation of static and dynamic Web servers that run on a bare PC is described in [6]. We only describe the additional design and implementation details that are pertinent to splitting the HTTP request and the TCP connection. As mentioned before, the TCB entry stores all the state information needed for each request in the split servers. The state information captures all the information needed by a bare PC server to process a request. When the state is transferred to another server, this is not a process migration [11] since there is no process attributed to the request. The state in the TCB entry captures all the information related to the client request, and also the state of the request as indicated in the state transition diagram (STD) for the HTTP request. The TCP code written for bare PC is a re-entrant code that does not use any local variables. Thus, the STD drives the server engine to process client requests independent of any information related to a given executing node. This approach is unique to bare PC applications and provides a simple mechanism to move requests from one bare PC server to another by simply sending the TCB entry to the other server. This paper only addresses delegate messages sent from CS to DS to achieve and support the splitting. However, although not discussed here, the same mechanism can be used for load balancing in a distributed system.

The original bare PC Web server design has three task types: a main task, a receive task, and the HTTP task for each request. We found that having a separate HTTP task for each request limits the maximum number of requests processed by the Web server. We have changed this design to facilitate splitting by eliminating the HTTP tasks. Thus, splitting requires having only the main and receive tasks. This approach resulted in better throughput for the server as it reduces task overhead. However, under high load conditions, the response time for the client increases as the receive task handles the entire request. We do not investigate the relationship between task design and response time in this paper.

In the new design for splitting, the main task gets control when a PC is booted and it executes a continuous loop. Whenever a new packet arrives or a timer expires, the main task invokes the receive task. The receive task processes the incoming packet and updates the TCB entry status. It invokes a procedure that handles packet processing. This entire process is executed as a single thread of execution (no threads in the system) without any other interrupts except for hardware interrupts including timer, keyboard, and NIC transmitter. This simple server design avoids all complexity that exists in an OS or kernel-based system.

The CS can make its decision on whether to split an HTTP request and TCP connection based on several factors as mentioned before. When it makes a decision to split, it will assemble DM1 and send it to its DS. The DM1 (and other DMs) can be sent over UDP or directly as a special message over IP since this message is between bare PC servers. Alternatively, a permanent TCP connection could be established between CS and DS to send DMs. However, this will increase splitting overhead. DM packets are sent from CS (master) to DS (slave). The CS will process the

GET message and do appropriate checking before sending the DM packet. Similarly, the CS will also check the data ACKs and FIN-ACKs, and monitor the state of data transmission to send DM packets when retransmissions are needed. In a LAN environment, retransmissions are rare.

The BMC Web server design for split operations is simple and extensible. It does not have any unnecessary functionality or overhead due to any other software running in the machine. The Web server application object (AO) is the only code running in the bare PC. The AO has complete control of the machine and communicates to hardware directly from its application program without using any interrupts.

The implementation of split servers is done using C/C++ code, and the code sizes are similar to our previous designs [5]. The STD approach is used to implement the HTTP protocol and the necessary network protocols. The bare PC Web server runs on any Intel-based CPUs that are IA32 compatible. It does not use any hard disk, but used BIOS to boot the system. A USB is used to boot, load, and store a bare PC file system (raw files at this point). There was no need to change any hardware interface code during Web server modification to handle splitting.

The software is placed on the USB and includes the boot program, startup menu, AO executable, and the persistent file system (used for resource files). The USB containing this information is generated by a tool (designed and run on MS-Windows) that creates the bootable Bare PC application for deployment.

V. PERFORMANCE MEASUREMENTS

A. Experimental Setup

The experimental setup involved Dell Optiplex GX260 PCs with Intel Pentium 4, 2.8GHz Processor, 1GB RAM and Intel 1G NIC on the motherboard. A LAN is set up for the experiments using a Linksys 16 port GB switch. Linux clients are used to run the http_load [17] stress tool and a bare PC Web client. Each http_load stress tool can run up to 1000 concurrent HTTP requests per sec. Each bare PC Web client can run up to 5700 HTTP requests per sec. A mixture of bare and Linux clients along with bare split servers are used to measure the performance. In addition, we tested the split servers with popular browsers running on Windows and Linux (Internet Explorer and Firefox respectively).

B. Measurements and Analysis

Internal Timing Comparison: Figure 3 shows the HTTP protocol timing results including the TCP interactions for non-split and split servers. A client request is issued to S1, which acts as a CS, and it can delegate the request to S2, which is the DS. The client request involves a resource file size of 4K. A Wireshark packet analyzer was used to capture and measure timings. The results were used to determine the latency overhead involved in splitting. The typical latency measured between GET-ACK and Header data is about 20 microseconds without splitting and 78 microseconds with splitting. That is, the split message and delegate server latency result in about 58 microseconds additional delay. The network overhead is two packets: one DM1 message and one DM2 message (168 bytes each). The split and non-split servers have the same behavior except for the additional latency mentioned above that contributes to the delay at the DS in sending the header.

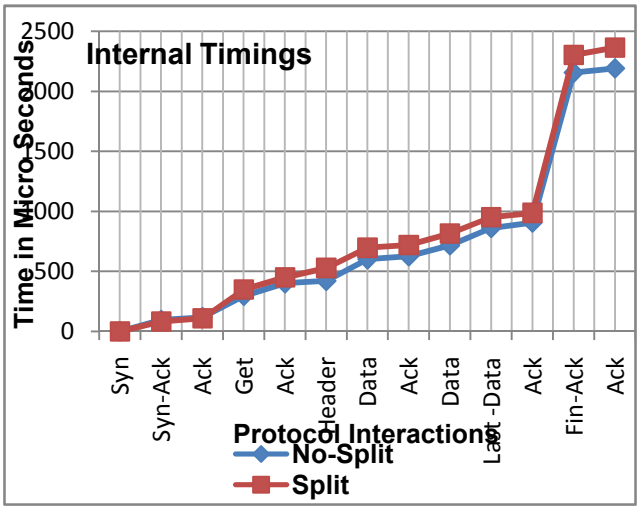

Figure 3. Internal Timings for HTTP/TCP

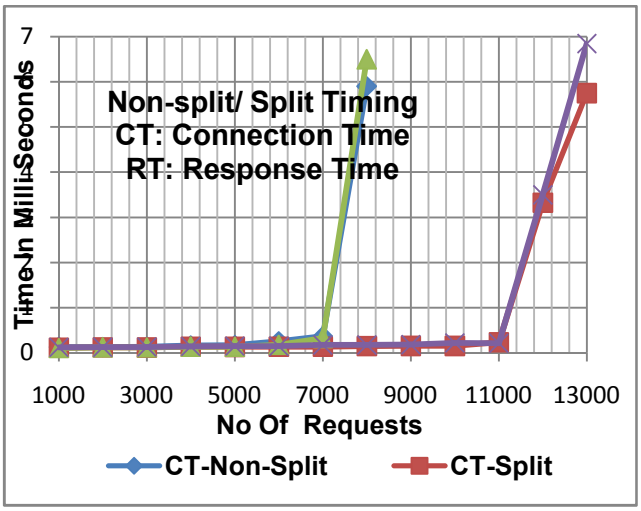

Figure 4. Response/Connection Times

Response/Connection Times: Figure 4 shows the response time and connection time at the client for varying HTTP request rates. S1 splits 50% of requests (with 0.5 probability) and delegates them to S2. Similarly, S2 splits 50% of its requests and delegates them to S1. For a 4K resource file size, the non-split server can handle up to 8000 requests/sec. Note that the connection and response times are similar. However, these times start increasing rapidly at 5500 requests/sec for the non-split server until it reaches its peak performance at 8000 requests/sec. For split servers, similar behavior is exhibited by a rapid increase starting at 11000 requests/sec and reaching a peak at 13000

requests/sec. The split servers provided a performance scale up of (13000/8000 = 1.625). However, we found that this is not the maximum capacity of split servers as shown later. The split server approach also demonstrates that it can be used for load balancing. The increase in response time and connection times are due to accumulating a large number of requests under heavy load conditions, and also due to the high CPU utilization as discussed below.

CPU Utilization: Figure 5 shows CPU utilization with respect to load variations for split and non-split cases. The CPU utilization reaches 93% for a non-split server and 92% for the split servers.

The main and receive tasks together take 100% CPU time (93% for the receive task and 7% for the main task). When 50% requests are sent from one server to other in both directions, the maximum capacity of both servers is reached at 13000 requests/sec.

Varying Split Percentage One-way: A non-split server provides a maximum capacity of 8000 requests/sec. When a split server is added, S1 (CS) can split requests and delegate them to S2 (DS). This is one-way from S1 to S2 only. That is, the S1 server is handling connections and data, and the DS server is only handling data transfers. We vary the split percentage at S1 and measure the maximum capacity of this server as shown in Figure 6. Note that the maximum capacity of S1 server is 12000 requests/sec. Thus, the S1's capacity is increased by 50% (12000 instead of 8000 requests/sec) by adding a DS. The CPU utilization chart for S1 and S2 is shown in Figure 7. The CPU utilization for S1 did not decrease rapidly as it is handling the maximum number of connections in all cases in addition to serving data. When it is not serving data at all (split 100%), it shows a drop in CPU utilization of about 5%. Also, when S1 acts as a data server it is not consuming much CPU time as the sending of data is also part of the receive task. As the number of split requests increases, the S2 server is utilized more, and eventually reaches close to saturation as well. The CPU utilization for S1 is 90% and for S2 is 86% at 100% split. That means, S1 is saturated, but S2 can handle more requests as it is only doing data transfers. In this situation, the data server (DS) is not saturated as much as the CS. This also implies that the work done by the data server is about 1000 requests/sec less than the connection server (if both servers share the load equally then the capacity of two servers would have been 13000 instead of 12000 requests/sec). In other words, a pure data server has 8.33% more capacity than the connection server (1000/12000).

Varying Split Percentage Both Directions: As seen above, varying the split percentage on server S1 increases server capacity to 12000 requests/sec. Similarly, when the split percentage is kept constant at 50%, the split server capacity increases up to 13000 requests/sec as shown in Figure 4. It is apparent that the optimal split server capacity should depend on the split percentage and its effect on server overhead. Thus, we continued the experiments to find the maximum server capacity by varying split percentage in both directions. Figure 8 show the chart for number of requests successfully processed by varying the split percentage. As there are two servers involved here, if each server's maximum capacity is 8000 requests/sec as shown in Figure 4, the theoretical maximum capacity of the servers should be 16000 requests/sec. The measurements indicate that the optimal split server capacity occurs at 25% (i.e., 25% requests are split by S1 and S2). We measured a combined capacity of 14,428 requests/sec when 25% of requests are split. That is, the scalability of split approach is 1.8035 for two servers (maximum can be 2.0). Thus, the two-server system loses only 20% capacity (10% for each server).

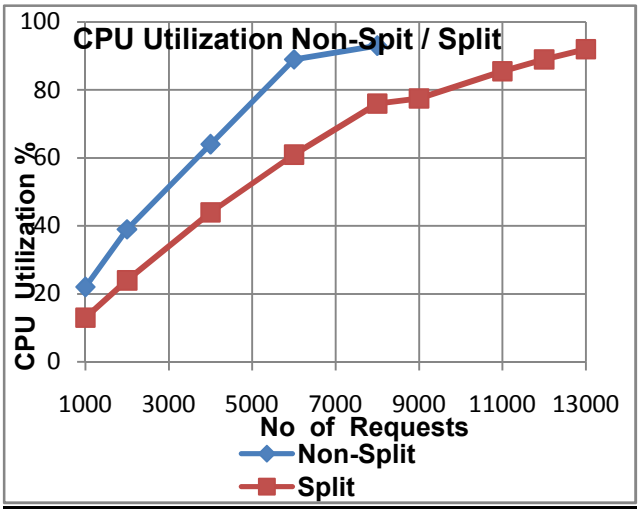

Figure 5. CPU Utilization

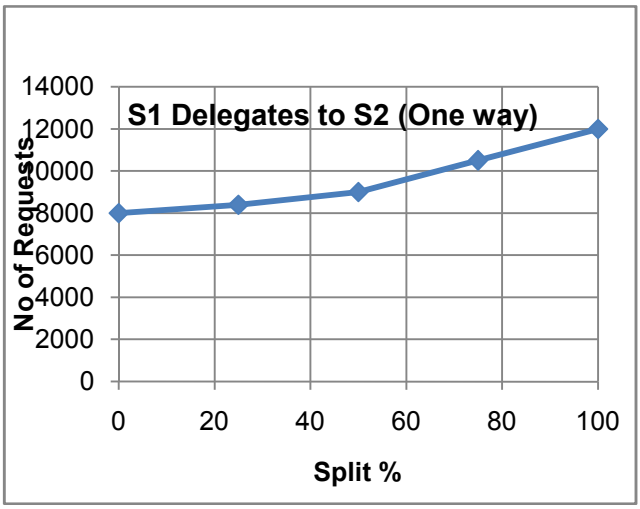

Figure 6. S1 Delegates to S2

It is evident that the overhead due to splitting operations is about 10% in this system. The CPU utilization for S1 and S2 are very close to each other, and the range varies between 95% through 89%. Figure 9 shows the CPU utilization with respect to varying split percentage. At 25% split, the CPU utilization is 92% (which is not the

maximum CPU utilization). The CPU utilization drops as the split percentage increases, because the data transfer is reduced at the server. However, when the split percentage is 100%, the utilization goes back up due to sending and receiving delegate messages.

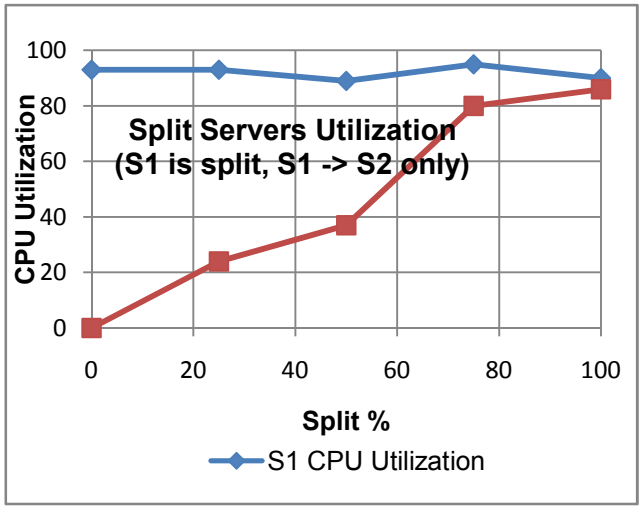

Figure 7. Slit Server Utilization

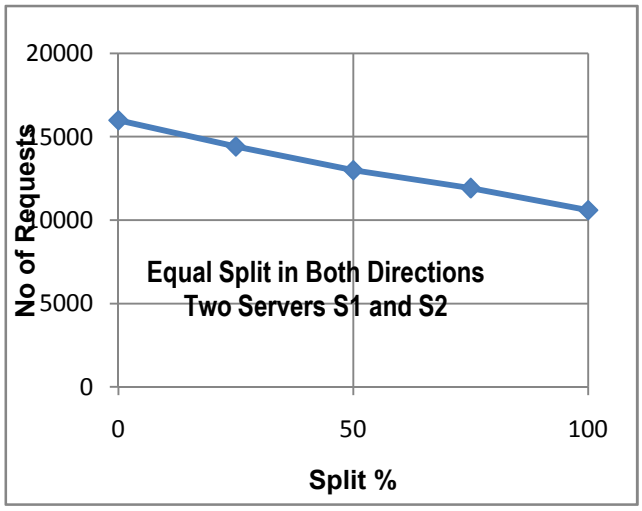

Figure 8. Equal Split in Servers

Varying Resource File Size: We varied the resource file size from 4K bytes to 32K bytes to determine its effect on splitting. Figure 10 shows the maximum number requests successfully processed with and without splitting. In this experiment, S1 and S2 get an equal number of requests, and 25% of the requests are split. The splitting percentage of 25% is used as it maximizes split server capacity. The results indicate that as the file size increases, the maximum capacity of the server to handle requests drops dramatically. This behavior is expected as the HTTP request processing time as shown in Figure 1 increases due to processing a large number of packets. The NIC transmitter also gets very busy and saturated while handling large number of packets. Figure 11 shows the corresponding processor utilizations for S1 and S2 servers. Notice that the utilizations drop as file size increases (since there are a fewer number of requests).

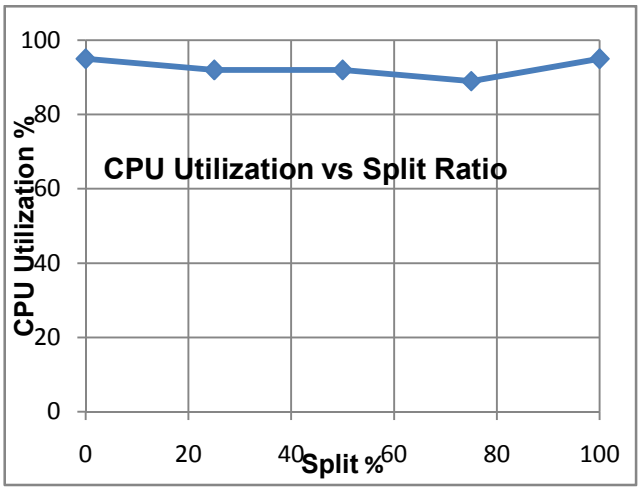

Figure 9. Varying Split Ratio on Both Servers

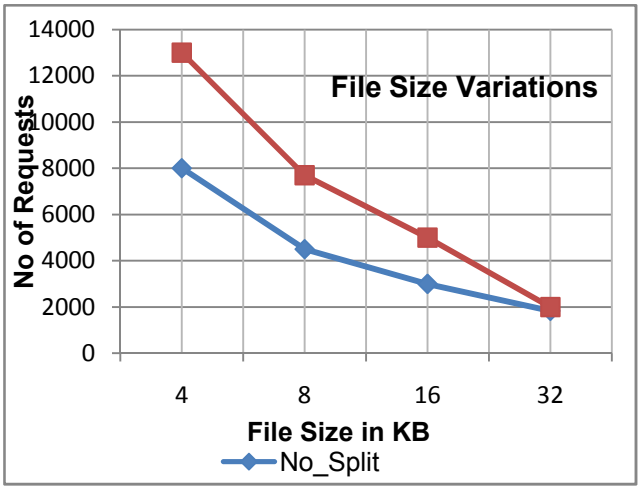

Figure 10. File Size Variations

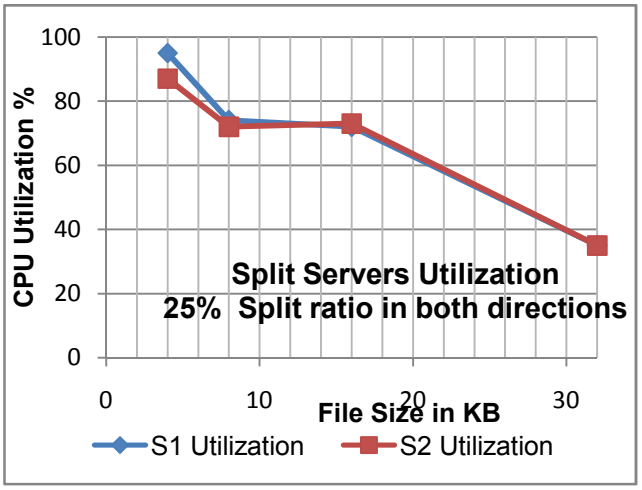

Figure 11. CPU Utilization 25% Split

Discussion of Results: Our empirical results demonstrate that splitting is feasible in a LAN environment and can be used for load sharing without any client involvement or central control. The split server scalability as shown (up to 90%) will enable us to develop a scalable BMC cluster architecture that can provide reliable service for clients. The architecture can also support the concept of delegating requests to other servers (for example to servers storing the data). The splitting concept can also be used in cases where one server (connection server) can be a master, and another server (data server) can be a slave. Server load balancing can be done based on splitting HTTP requests and the underlying TCP requests instead of dispatching requests to other nodes.

VI. CONCLUSION

In this paper, we showed how to split an HTTP request and the underlying TCP connection between the two servers. We presented the split architecture and provided the relevant design and implementation details. However, the idea is demonstrated using BMC (bare PC) servers as splitting would be much harder to implement using OS-based servers. The experimental results indicate that when HTTP requests are split using two servers and delegation is done in both directions (25% delegation to each other), the maximum capacity is 14,428 requests/sec (the theoretical limit is 16,000 for two servers). Splitting scalability was found to be 1.8035 for two servers (i.e., there is an approximately 10% overhead due to splitting). With respect to increasing the file size, it was seen that performance is similar with or without splitting. Moreover, the splitting percentage does not have much impact for larger file sizes. The novel splitting technique and associated Web server architecture introduced in this paper have potential applications in distributed computing and improving server reliability.

REFERENCES

[1] G. Ammons, J. Appayoo, M. Butrico, D. Silva, D. Grove, K. Kawachiva, O. Krieger, B. Rosenburg, E. Hensbergen, R.W. isniewski, "Libra: A Library Operating System for a JVM in a Virtualized Execution Environment," VEE '07: Proceedings of the 3rd International Conference on Virtual Execution Environments, June 2007.

[2] G. Canfora, G. Di Santo, G. Venturi, E. Zimeo and M.V.Zito, "Migrating web application sessions in mobile computing," Proceedings of the 14th International Conference on the World Wide Web, 2005, pp. 1166-1167.

[3] D. R. Engler and M.F. Kaashoek, "Exterminate all operating system abstractions," Fifth Workshop on Hot Topics in operating Systems, USENIX, Orcas Island, WA, May 1995, p. 78.

[4] G. R. Ganger, D. R. Engler, M. F. Kaashoek, H. M. Briceno, R. Hunt and T. Pinckney, "Fast and flexible application-level networking on exokernel system," ACM Transactions on Computer Systems (TOCS), Volume 20, Issue 1, pp. 49 – 83, February, 2002.

[5] L. He, R. K. Karne, and A. L. Wijesinha, "The Design and Performance of a Bare PC Web Server," International Journal of Computers and Their Applications, IJCA, Vol. 15, No. 2, June 2008, pp. 100-112.

[6] L. He, R.K. Karne, A.L Wijesinha, and A. Emdadi, "A Study of Bare PC Web Server Performance for Workloads with Dynamic and Static Content," The 11th IEEE International Conference on High Performance Computing and Communications (HPCC-09), Seoul, Korea, June 2009, pp. 494-499.

[7] R. K. Karne, K. V. Jaganathan, and T. Ahmed, "DOSC: Dispersed Operating System Computing," OOPSLA '05, 20th Annual ACM Conference on Object Oriented Programming, Systems, Languages, and Applications, Onward Track, ACM, San Diego, CA, October 2005, pp. 55-61.

[8] R. K. Karne, K. V. Jaganathan, and T. Ahmed, "How to run C++ Applications on a bare PC," SNPD 2005, Proceedings of NPD 2005, 6th ACIS International Conference, IEEE, May 2005, pp. 50-55.

[9] R. K. Karne, "Application-oriented Object Architecture: A Revolutionary Approach," 6th International Conference, HPC Asia 2002 (Poster), Centre for Development of Advanced Computing, Bangalore, Karnataka, India, December 2002.

[10] J. Lange, K. Pedretti, T. Hudson, P. Dinda, Z. Cui, L. Xia, P. Bridges, A. Gocke, S. Jaconette, M. Levenhagen, R. Brightwell, "Palacios and Kitten: New High Performance Operating Systems for Scalable Virtualized and Native Supercomputing," Proceedings of the 24th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2010), April, 2010.

[11] D.S. Milojicic, F. Douglis, Y. Paindaveine, R. Wheeler and S. Zhou. "Process Migration," ACM Computing Surveys, Vol. 32, Issue 3, September 2000, pp. 241-299.

[12] V. S. Pai, P. Druschel, and Zwaenepoel. "IO-Lite: A Unified I/O Buffering and Caching System," *ACM Transactions on Computer Systems,* Vol.18 (1), ACM, February 2000, pp. 37-66.

[13] K. Sultan, D. Srinivasan, D. Iyer and L. lftod. "Migratory TCP: Highly Available Internet Services using Connection Migration," Proceedings of the 22nd International Conference on Distributed Computing Systems, July 2002.

[14] T. Venton, M. Miller, R. Kalla, and A. Blanchard, "A Linux-based tool for hardware bring up, Linux development, and manufacturing," *IBM Systems J.*, Vol. 44 (2), IBM, NY, 2005, pp. 319-330.

[15] D. Wentzlaff and A. Agarwal, "Factored operating systems (fos): the case for a scalable operating system for multicores," ACM SIGOPS Operating Systems Review, Volume 43, Issue 2, pp. 76-85, April 2009.

[16] D. Zagorodnov, K. Marzullo, L. Alvisi and T.C. Bressourd, "Practical and low overhead masking of failures of TCP-based servers," ACM Transactions on Computer Systems, Volume 27, Issue 2, Article 4, May 2009.

[17] http://www.acme.com/software/http_load.

[18] http://www.tinyos.net/.

[19] A. Cohen, S. Rangarajan, and H. Slye, "On the performance of TCP splicing for URL-Aware redirection," Proceedings of USITS'99, The 2nd USENIX Symposium on Internet Technologies & Systems, October 1999.

[20] Y. Jiao and W. Wang, "Design and implementation of load balancing of a distributed-system-based Web server," 3rd International Symposium on Electronic Commerce and Security (ISECS), pp. 337-342, July 2010.