

TOWSON UNIVERSITY
COLLEGE OF GRADUATE STUDIES AND RESEARCH

SPLIT PROTOCOLS IN BARE MACHINE COMPUTING

by

Bharat Singh Rawal Kshatriya

A Dissertation

Presented to the faculty of

Towson University

in partial fulfillment of

the requirements for the degree of

Doctor of Science in Information Technology

December 2011

Towson University
Towson, Maryland 21252

© 2011 Bharat Singh Rawal Kshatriya

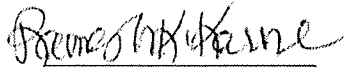
All Rights Reserved

TOWSON UNIVERSITY
COLLEGE OF GRADUATE STUDIES AND RESEARCH

DISSERTATION APPROVAL PAGE

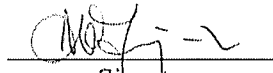
This is to certify that the thesis prepared by Bharat Singh Rawal Kshatriya entitled "SPLIT PROTOCOLS IN BARE MACHINE COMPUTING" has been approved by this committee as satisfactory completion of the requirement for the degree of Doctor of Science in Information Technology.

Dr. Ramesh K. Karne
Chair, Thesis Committee


Signature

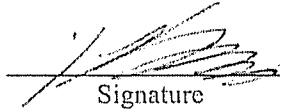
11-15-2011
Date

Dr. Alexander L. Wijesinha
Committee Member


Signature

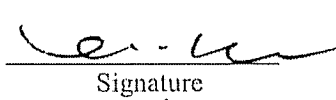
11/15/2011
Date

Dr. Yeong-Tae Song
Committee Member


Signature

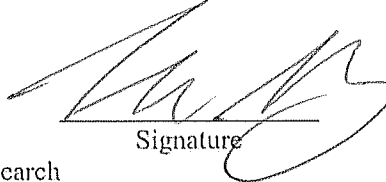
11/21/2011
Date

Dr. Wei Yu
Committee Member


Signature

11/21/2011
Date

Dr. Lawrence Shirley
Associate Dean,
College of Graduate Studies and Research


Signature

22 Nov 2011
Date

Acknowledgements

I would like to express my appreciation to all those who have supported my efforts to complete this dissertation. I am greatly appreciative of my research committee Dr. Ramesh K. Karne (chair), Dr. Alexander Wijesinha, Dr. Yeong-Tae Song, and Dr. Wei Yu for supporting this research. I am especially thankful to Dr. Karne and Dr. Wijesinha for all the long hours in lab, and support and advice I have received throughout this dissertation research. Also I am thankful to my mother Bashundhara Rawal, wife Laxmi Rawal, son Atul Rawal, daughter Sima Rawal, my relatives Anni Rawal and Sharswati Khadaka, who have heartily supported my academic goals during long period of my doctoral study.

I also would like to thank Dr. Chao Lu, Chair of the Department of Computer and Information Sciences, and Dr. Lawrence Shirley the Acting Dean of College of Graduate Studies and Research at Towson University, for facilitating this work. Gratitude is also given to the late Frank Anger (National Science Foundation) for his support of the Application oriented Object Architecture, which evolved into Bare Machine Computing research and this dissertation.

ABSTRACT

SPLIT PROTOCOLS IN BARE MACHINE COMPUTING

Bharat Singh Rawal Kshatriya

This thesis extends on-going Bare Machine Computing (BMC) research at Towson University. BMC applications run on a bare machine without any commercial operating system, kernel, or other centralized support. This research consists of splitting protocols and their impact on performance, its applicability to servers and clients, ability to construct scalable mini or large server clusters. The split protocol principle discovered in this dissertation laid a basic foundation for protocol splitting and demonstrated unexpected performance improvements in constructing split protocol based servers. This dissertation will also serve as a cornerstone for future split protocol architectures, design and implementation for all OS based systems.

The split protocols found and investigated in this thesis fall into two main categories. First, a protocol may be split based on a connection and data interactions. One server handles connection establishment and closing and the other handles data for each transaction. Both servers stay online until a given request is complete. A given client does not see such splitting during its transactions. We refer to this approach as a server level split protocol. Second, a protocol may be split such that a connection server will provide connection establishment and a data server will provide data and termination of the connection. In this case, the client is fully aware of connection and data servers. This approach is referred to as a split protocol at client server architecture level. The later technique modifies the existing client server protocol and makes the client aware of splitting protocols. Both of these approaches have pros and cons of their usage. This thesis investigates these two approaches in detail and derives numerous key impacts of protocol splitting.

The two approaches outlined above are demonstrated using a bare PC implementation that is conformed to a Bare Machine Computing paradigm invented by Dr. Karne at Towson University. Numerous performance measurements are conducted during the research and it was found that two split servers perform about 25% more than a conventional two independent servers. This is the most surprising result found in this research, where such dramatic improvements in performance in Web servers were never discovered before. In addition, this work demonstrated an efficient way to construct mini clusters ranging up to 4 servers and large clusters ranging up to 16 servers. This dissertation work will also serve as a cornerstone for building scalable server clusters based on split

protocol discovery. The Split Protocol concept explored here will provide a generic idea to split transactions in general to achieve higher performance in distributed servers on the network. Numerous applications such as file servers, database servers and webmail servers may benefit from Split Protocol concept. Finally, the split protocol architecture may also result in more reliable server clusters due to its redundant nature of dual servers having complete state of each request in processing.

Table of Contents

List of tables	ix
List of figures.....	x
1. MOTIVATION	13
2. RELATED WORK	14
2.1 Bare Machine Computing Background.....	14
2.2 Bare Machine Computing Applications.....	16
2.3 Other Related Work	16
3. Introduction.....	18
4. insight into bare pc web server	23
4.1 Background of Bare PC Web server	23
4.2 Web Server Architecture.....	24
4.3 Design and Implementation	26
5. SPLIT SERVERS	33
5.1 Split Protocol Architecture.....	33
5.2 Design and Implementation	37
5.3 Performance Measurements	40
5.3.1 Experimental Setup.....	40
5.3.2 Measurements and Analysis.....	41
5.3.2.1 Internal Timing Comparison	41
5.3.2.2 Response/Connection Times.....	42
5.3.2.3 CPU Utilization.....	42
5.3.2.4 Varying Split Percentage One-way	42
5.3.2.5 Varying Split Percentage Both Directions	43
5.3.2.6 Varying Resource File Size.....	47
6. MINI-CLUSTER CONFIGURATION	50
6.1 Cluster Configurations	50
6.2 Performance Measurements	53
6.2.1 Configuration 1 (1 CS, 1 DS, full delegation)	53
6.2.2 Configuration 2 (1 CS, 1-3 DS, full delegation)	55
6.2.3 Configuration 2 (1 CS, 1-3 DS, partial delegation)	56
6.2.4 Configuration 3 (2 CS, 1 DS, partial delegation).....	59
7. MODIFIED CLIENT/SERVER ARCHITECTURE	62
7.1 Background	62
7.2 Design and Implementation	64
7.3 Experiment Results	66
7.3.1 Experimental Setup.....	66
7.3.2 Configurations.....	66
7.3.3 Measurements: 1-4 DSs, Performance	67
7.3.4 Measurements: 1-4 DSs, Actual Times	69
7.3.5 Lack of comparison with OS based systems	72
7.4 Impacts of Modified Splitting Protocol	72

8.	SIGNIFICANT CONTRIBUTIONS	75
9.	CONCLUSION.....	77
	Appendices	79
	References	83
	Curriculum Vitae.....	93

LIST OF TABLES

Table 1. Data for figure 3.....	79
Table 2. Data for figure 4.....	79
Table 3. Data for figure 5.....	80
Table 4. Data for figure 6.....	80
Table 5. Data for figure 8.....	80
Table 6. Data for figure 9.....	81
Table 7. Data for figure 10.....	81
Table 8. Data for figure 11.....	82
Table 9. Data for figure 12.....	82
Table 10. Data for figure 13.....	82
Table 11. Data for figure 14.....	82
Table 12. Data for figure 15.....	83
Table 13. Data for figure 16.....	83
Table 14. Data for figure 20.....	83
Table 15. Data for figure 21.....	84
Table 16. Data for figure 22.....	84
Table 17. Data for figure 23.....	84
Table 18. Data for figure 24.....	85
Table 19. Data for figure 25.....	85
Table 20. Data for figure 26.....	85
Table 21. Data for figure 27.....	85
Table 22. Data for figure 28.....	86
Table 23. Data for figure 29.....	86
Table 24. Data for figure 30.....	86
Table 25. Data for figure 34.....	86
Table 26. Data for figure 35.....	87
Table 27. Data for figure 36.....	87
Table 28. Data for figure 37.....	87
Table 29. Data for figure 38.....	88

LIST OF FIGURES

Figure 1. Client/Server Interactions.....	20
Figure 2. Architecture of the Bare PC Web Server.....	27
Figure 3. No. of Connections versus Processing Time.....	29
Figure 4. No. of Connections versus Task Utilizations.....	30
Figure 5. No. of Connections vs Max Queue Size.....	31
Figure 6. HTTP/RCV Task Utilization Plot.....	32
Figure 7. Split Architecture.....	35
Figure 8. Internal Timings for HTTP/TCP.....	40
Figure 9. Response/Connection Times.....	41
Figure 10. CPU Utilization.....	44
Figure 11. Split % (S1 Delegates to S2).....	44
Figure 12. Split Server Utilization.....	45
Figure 13. Equal Split in Servers.....	45
Figure 14. Varying Split Ratio on Both Servers.....	46
Figure 15. File Size Variations.....	47
Figure 16. CPU Utilization 25% Split.....	48
Figure 17. Split architecture configuration 1.....	51
Figure 18. Split architecture configuration 2.....	52
Figure 19. Split architecture configuration 3.....	52
Figure 20. Throughput with increasing file sizes (Configuration 1).....	53
Figure 21. CPU utilization with increasing file sizes (Configuration 1).....	54
Figure 22. Connection and response times (Configuration 1, file size 64KB).....	54
Figure 23. DS throughput (Configuration 2, file size 64KB).....	55
Figure 24. Connection and response times (Configuration 2, file size 64K).....	56
Figure 25. CPU Utilization (Configuration 2, file size 64KB).....	57
Figure 26. Throughput with full/partial delegation (Configuration 2, file size 64KB).....	58
Figure 27. Throughput with full/partial delegation for varying file sizes (Configuration 2).....	58
Figure 28. Connection and response times (Configuration 2, file size 64KB).....	59
Figure 29. Throughput with full/partial delegation (Configuration 3, file size 4KB).....	60
Figure 30. Connection times and response times with full/partial delegation (Configuration 3, 4KB).....	60
Figure 31. Split Protocol Architecture	63
Figure 32. Design Structure.....	65
Figure 33. Split Architecture Configurations.....	67
Figure 34. Varying DSs, Performance Chart	68
Figure 35. Varying DSs, CPU Utilization... ..	68
Figure 36. Varying DSs, Actual Time.....	69
Figure 37. Varying ACKs, Performance Chart.....	70
Figure 38. Varying ACKs, Utilizations.....	71

1. MOTIVATION

A protocol assumes a continuous communication between two communicating entities. It is also a set of interactions between these two entities. This implies that these two entities are tightly coupled to process a given transaction and communicate in a request response manner to finish the transaction. This is a very general way to describe a protocol. However, most commonly used protocol on the Internet is a client server protocol. This in turn consists of some high level protocols such as HTTP and some low level protocols such as TCP and so on. Similarly, there are many other protocols such as FTP, TLS, SMTP, POP3, and so on. In general, most of these protocols seem to conform to a generic principle that a given protocol is intact or *in-separable* and defines a rigid interaction between two communicating entities.

The in-separable characteristics of protocols motivated us to develop a split protocol where the conventional wisdom of intactness is broken. A protocol can be broken in many ways as illustrated in this dissertation. The split protocol concept resulted in surprising results and paved the way into unknown territories resulting in novel characteristics that will be useful to build future client server technologies and clusters.

2. RELATED WORK

2.1 *Bare Machine Computing Background*

Bare Machine Computing (BMC) first invented by Dr. Karne at Towson University is motivated by the unique concept to make a computing box bare and carry the software application in a portable device such as a flash-drive. Bare machine applications use the BMC or dispersed operating system concept [28]. That is, there is no operating system (OS) or centralized kernel running in the machine. Instead, the application is written in C++ and runs as an application object (AO) [27] by using its own interfaces to the hardware and device drivers.

When computing hardware is made bare, the bare hardware can be used to run any given application on the fly. There is no need to protect the bare box as it does not have any valuable resources. The bare box simply has memory, CPU, user interface (input/output) and a network interface. All persistent data is stored externally on a mass storage device or on the network. This BMC approach is applicable to any pervasive device including: desktop, laptop, hand-held, or any other electronic equipment.

The BMC concept is not an embedded approach as it is applicable to generic computing. The software application(s) can be modeled as a single monolithic executable as an application object (AO), which is based on a single end-user application or a set of end-user applications that are required at a given time. For example, a Web browsing is an end-user application. One can design an AO that consists of simply a Web browser, or a composite of applications including: Web browser, Web Mail, E-Transaction and Text processing. The computers carry their applications as AOs on a flash-drive. In this computing, the information technology world is application centric rather than environment dependent as AOs can be run on any bare machines.

One can develop AOs based on end-user applications and they can be made tailored to suit individual needs. The AO is written in a single programming language and compiled into a given bare machine code. These AOs can be run on any bare computers which have no particular ownership. For example, one can walk into an organization and use their computer by using their own AO, without harming the bare computer. This computing paradigm will change the way we do business today and make the computing devices standard and universal. The AOs can be open ended application domains. The computer programming languages will become standard and extensible as the applications grow. There will be a greater need for standardizing hardware, software and interfaces for bare machines.

The bare machine computing applications are small, end-user centric and application centric. Once an AO programmer is trained, it is easy to write AOs, as it requires a single programming language expertise and only the AOs domain knowledge, and there is no need to know other computing platforms or environments. The AO programmer is in total control of a given application set's design and its execution order. No centralized OS or kernel is involved in its execution. As the AO controls the application aspect as well as the execution aspect; these applications will be inherently more secure. The AOs are quite suitable for peer-to-peer secure communications. In fact, the bare to bare communication reaches the ultimate security one can achieve in peer-to-peer communication, as it avoids all the system related vulnerabilities by making the device bare. The AOs can be tailored on the fly for a given set of users or group of users. The communication and security protocols can also be tailored to provide more secure communications. The computing aspect of bare machines becomes standard and limited, but the network and security aspect of computing becomes more open for efficient protocols that are suitable to end users.

2.2 Bare Machine Computing Applications

The feasibility of BMC paradigm has been demonstrated by building complex applications in the bare machine computing laboratory at Towson University. Several doctoral dissertations have been completed in this area. Long He [22] developed the first bare PC Web server and demonstrated the feasibility of building complex software that runs on a bare PC with thousands of threads and outperforms other compatible commercial Web servers. Gholam Khaksari [29] developed first VoIP soft-phone that runs on a bare PC and provides a secure communication on an end-to-end basis. Andre Alexander [1] built a SIP server and a bare SIP agent to demonstrate the feasibility of high performance SIP server with secure communication using SRTP protocol. George H. Ford built first Email server that runs on a bare PC and provides compatible performance related commercial email servers [20]. Ali Emdadi [11] implemented the TLS protocol on top of bare Web server. TLS is a complex protocol and requires numerous security algorithms that run on bare PC. Roman Yasinovskyy [48] implemented IPv6 protocol that runs on a bare PC. These doctoral works discovered many novel characteristics that are unique to BMC.

2.3 Other Related Work

Since the inception of large mainframe computers; the software complexity has been growing exponentially. The enormous growth in computing hardware and software has created unmanageable electronic dump [31] without any reuse of hardware and software. There are over 30 versions of OS releases in last 25 years from Microsoft Corporation alone. Software applications, operating systems, tools and gadgets come and go on a daily basis without serving their useful life cycle. The OS code sizes have reached close to one hundred million lines of code resulting in rapid upgrades, version releases, errors and security flaws.

Many researchers now focus on coping with small kernels and lean operating systems or dedicated applications. While the BMC concept resembles approaches that reduce OS overhead and/or use lean kernels such as OS abstraction [15,16,17], Exokernel OS [8, 9, 14], Also in IO-

Lite [38], Palacio [33], Libra [3], bare-metal Linux [45], OS Kit[44], Factored OS [47] and TinyOS [43], Fast and flexible networking [21], there are significant differences such as the lack of a centralized code that manages system resources and the absence of a standard TCP/IP protocol stack. In essence, the AO itself manages the CPU and memory, and contains lean versions of the necessary protocols. Protocol intertwining is a form of cross-layer design. Further details on bare PC applications and bare machine computing (BMC) can be found in [18, 19].

Splitting protocol at a client server architecture level is different from migrating TCP connections, processes or Web sessions; splicing TCP connections; or masking failures in TCP-based servers. For example, in migratory TCP (M-TCP) [42], a TCP connection is migrated between servers with client involvement. In process migration [6], an executing process is transferred between machines. In proxy-based session handoff [35], a proxy is used to migrate web sessions in a mobile environment. In TCP splicing [10], two separate TCP connections are established for each request. In fault-tolerant TCP (FT-TCP) [49], a TCP connection continues after a failure enabling a replicated service to survive. Per our knowledge, no work on splitting protocol connections at client server architectural level has been done before.

3. INTRODUCTION

A Web server serves static and dynamic content to clients. Its performance and reliability are essential and crucial to its survival in today's world. In addition, its maintenance and longevity are also essential to reduce its operating cost and management. Web servers are also focused products where they only perform certain functions and do not require much user interfaces. Thus, Web servers or servers in general are design-friendly to bare PC applications.

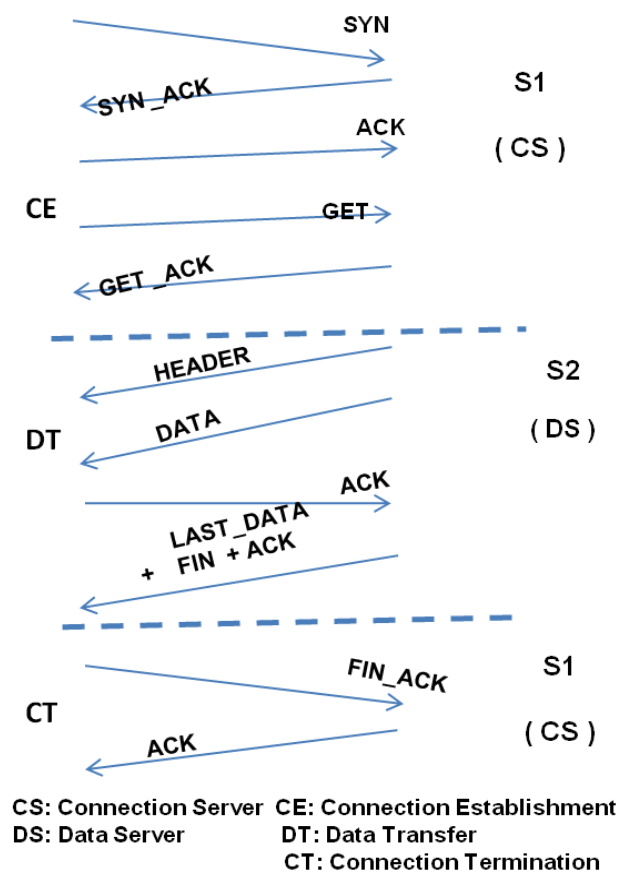
Considering the above characteristics of Web servers, it is obvious that there is no need to use a full-blown OS based servers that require periodic maintenance, updates, and prone to OS based attacks. Many alternatives to server design have been proposed in the literature. Google-cluster proposed its own lean Linux kernel [4]; SWILL [32] suggests a special Web server library to run their server applications. Exokernel [8] recommends to move network intensive applications from kernel to applications to gain high speed-up. Many server systems disable the usual OS functions such as shell scripts, logs, remote logins, and so on. Commercial servers such as Apache focus their designs to optimize for performance and provide API to hardware resources instead of system calls. Instead of trying all these approaches, we propose to avoid any OS, kernel, or any centralized software to manage resources and give the control back to the application programmer thus resulting in the BMC paradigm. The BMC approach is the extreme end of OS a spectrum where there is no OS related control running in the machine. When the machine is made bare, then it provides immense potential to develop applications that are independent of computing environments. It only depends upon the underlying CPU architecture. The application programmer job made more complex by making the programmer to deal with application and system's knowledge. However, the applications themselves are simple and easy to code once the programmer understands underlying principles of BMC.

The design and implementation of bare PC Web server was presented in [22]. The static and dynamic performance of this server was also presented in [23]. The direct hardware interfaces through an AP were described in [15].

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 [2], [10], and [16]. Alternatively, in approaches such as Migratory TCP (M-TCP) [42], 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 [23] that runs applications without an operating system (OS) or kernel. Protocol intertwining in BMC has been previously shown to improve Web server performance [22], [23]. The application-centric BMC Web server design and architecture [22] includes protocols as part of the application and facilitates inter-layer protocol communication.

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



No table of figures entries found.

Figure 1. HTTP/TCP Protocol Interactions

client during the data transfer. This process differs somewhat based on whether the HTTP request is static or dynamic. Although we address 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 here 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 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.

Load balancing is frequently used to enable Web servers to dynamically share the workload. For load balancing, a wide variety of clustering server techniques [25, 12, and 46] are employed. Most load balancing systems used in practice require a central control system such as a load balancing switch or dispatcher [12, and 25]. Load balancing can be implemented at various layers in the protocol stack [10], [25]. Various clustering approaches demonstrated for improving productiveness [5, 7]. We consider a new approach to load balancing that involves splitting HTTP requests among a set of two to four servers, where one or more connection servers (CSs) handle TCP connections and may delegate a fraction (or all) requests to one or more data servers (DSs) that serve the data [40]. For example, the data transfer of a large file could be assigned to a DS and the data transfer of a small file could be handled by the CS itself.

One advantage of splitting is that splitting systems are completely autonomous and do not require a central control system such as dispatcher or load balancer. Another advantage is that no client involvement is necessary as in migratory or M-TCP [42]. In [40], splitting using a single CS and a DS was shown to improve performance compared to non-split systems. Since the DSs

are completely anonymous and invisible (they use the IP address of the delegating CS), it would be harder for attackers to access them. In particular, communication between DSs and clients is only one-way, and DSs can be configured to only respond to inter-server packets from an authenticated CS. We study the performance of three different configurations of Web server clusters based on HTTP splitting by measuring the throughput (in requests/s) and also connection and response times at the client.

In real world applications, some servers may be close to data sources, and some servers may be close to clients. Splitting a client's HTTP request and the underlying TCP connection in this manner allows servers to dynamically balance the workload. We have tested the splitting concept in a LAN that consists of multiple subnets connected by routers. In HTTP splitting, clients can be located anywhere on the Internet. However, there are security and other issues that arise when deploying mini clusters in an Internet where a CS and a DS are on different networks [40].

4. INSIGHT INTO BARE PC WEB SERVER

Throughout this dissertation we have used BMC Web server as a baseline to conduct experiments and measurements for split protocol concept. Thus, it is essential to provide insight into the BMC server [39] in this thesis. Another reason for the justification of this section is also that the BMC server has been optimized for performance during this study. The optimized Web server is crucial to conduct performance measurements for scalable servers.

The following section presents an insight into a Bare PC Web server and describes the internal details. The novel architecture and design principles outlined here will serve as the future models of computation where systems can be designed for ultimate performance and optimization. This work will demonstrate a potential for building computer applications, which can inherently possess longevity thus reducing obsolescence. In addition, some new measurements for the Bare PC Web server such as CPU utilization, and maximum queue sizes are presented to demonstrate the potential of bare machine applications. Critical comparison of the Bare PC Web server as published before [22] is not the focus of this work.

4.1 *Background of Bare PC Web server*

The strength of BMC applications is derived from its simplicity, smaller code, design by obscurity, design for longevity, and inherent security. The BMC Web server is based on these principles. A Web server serves static and dynamic content to clients. Its performance and reliability are essential and crucial to its survival in today's world. In addition, its maintenance and longevity are also essential to reduce its operating cost and management. Web servers are also focused products where they only perform certain functions and do not require much user interfaces.

4.2 *Web Server Architecture*

The conventional Web servers devise intricate mechanisms [38] that either bypass OS or design special API to improve their performance. Sometimes, they use techniques such as Web caching [30], real memory instead of virtual memory, single monolithic executable (Linux), memory mapped files, and so on. The bare machine or bare PC applications do not have any OS and thus there is no need to devise any such mechanisms to achieve higher speeds. The novelty in BMC applications stems from its inherent characteristics. The BMC applications assume a standard underlying CPU architecture such as Intel X86 or similar. The applications directly communicate to this hardware. The boot, load, execute, multi-processing, exceptions, interrupts, error controls, memory management, file management, and I/O are part of the application object (AO). There will be direct interfaces to hardware [26], which will provide communication to hardware, which is keyboard, mouse, display, CPU, memory, and interrupts.

The direct hardware interfaces to the AO programmer is available at C/C++ programming level. These interfaces are not same as system libraries, which require some sort of underlying OS calls (e.g. INT 21H). The interfaces directly communicate with keyboard buffer, video memory, Ethernet card (NIC), interrupts through call gates or interrupt gates, tasking through TSS (task state segment), thus directly accessing hardware. The device driver for the NIC is also independent of any OS related controls or interrupts. The bare NIC driver allows the AO programmer to directly read UPD (upload) and DPD (download) buffers. In BMC applications all memory is real and dynamic memory is managed by the AO programmer and the memory map for the application is done a-priori with a required limit and allocations for a given set of applications.

The novelty of architectural features of the bare PC Web server as shown in Figure 2 can be described as follows. As Web server requests are independent of each other, each request is modeled as a separate task (which uses separate TSS) and these tasks do not share any data. The

task code can be shared by many tasks and thus these shared program variables are stored in a TCP table entry (called TCB entry). The concurrency control issues are solved by using a separate TCB entry for each request. In addition to the program variables, the TCB entry also carries some state variables for each request. The TCB entry in the bare PC Web server is about 200 bytes. This entry is independent of any particular machine or node that it resides; it can be easily transported to another node thus making the entry migratory.

There are a total of three types of tasks in the Web server. The Main task is always running which checks for any received packets. When a packet is received, it gives the control to a RCV task, where the received packet gets processed in a single thread of execution without any interruption and returns to Main task. When a GET request comes from a client, the RCV task inserts a HTTP task so that the HTTP request can be served. Main task and RCV tasks are independent entities and all HTTP tasks are placed in a single circular list to be processed first come first serve. When an HTTP task waits for an acknowledgement from a client, then it gets suspended and gets back into the circular list. When an ACK arrives from a client, then the suspended task will be resumed. The task structure and its implementation are tuned to serve only Web server requests. The architecture does not allow any other unrelated functionality in the Web server.

The TCB entry also keeps track of client's IP and Port Number as a unique hash entry into the table. All packets to be transmitted are placed in the UDP buffer once and tracked through a sliding window protocol for a required window size. Only transmit interrupts are used to confirm the transmitted packets.

The resource files for the Web server are transferred from a Microsoft Windows machine to the bare PC using trivial FTP. The files are stored in memory for access and updated as needed from a USB flash memory.

In summary, the architectural insight to the bare PC Web server is simple and designed for performance. The architecture is limited to its intended functionality. There are no open ports, no way to create unrelated tasks, and no way to get the control of CPU from its AO execution. The AO is a single monolithic executable created at compile time. There are no dynamic capabilities to modify or alter the program execution of the intended AO function. Thus, we believe that this Web server architecture is inherently secure and this model can be used to build any other servers in general.

4.3 Design and Implementation

The Web server design is focused around the intertwined HTTP client/server protocol as illustrated in Figure 1. For dynamic Web requests there will be a PHP protocol between the Web server and DB server [30] which is not shown in this thesis. A client request starts with a SYN packet which is received by the RCV task and an immediate ACK response is sent to the client. When an ACK comes from the client, then a connection is established, and it is noted in the corresponding TCB entry. When a GET arrives, the RCV task will insert an HTTP task as mentioned before and the HTTP task will handle the data transfer part of the protocol. When all the data is successfully sent then the server sends FIN-ACK and closes the connection. The protocol interaction as shown in Figure 2 is modeled as a state transition diagram (STD), and it is implemented as part of the HTTP task. The HTTP task and TCP protocol are intertwined to accomplish the total protocol. This intertwining of protocol is similar to cross-layering [40], instead of layers provides close to optimal in response time for the clients.

The design of the bare PC Web server consists of implementing the HTTP and PHP protocols and updating the TCB entries for each request. Main task, RCV task, and HTTP tasks together accomplish the total function required for processing client requests. Notice that simplicity of the design also stems from the single circular list and the first-come-first-serve priority approach. The simplicity is carried all over the design phase of bare PC Web servers.

The size of this assembly code is approximately 1,800 lines. These direct hardware interfaces include: display, keyboard, timers, task management, and real/protected mode switching. The 3COM 905CX NIC driver code is approximately 1400 lines of assembly code, with the rest of the code written in C. The implementation of the Web server architecture depicted in Figure 1 is written in C++ in an object-oriented fashion. The numbers of executable statements for this server are about 9000 not including comments. The resulting size of this single monolithic executable AO is 304 sectors of code (155,648 bytes).

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. The tool, which generates the boot load sector, and copies the executable and associated files to the USB, consists of only 469 lines of C++ code.

4.4 Performance Measurements

4.4.1 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 network is set up for the test using a Linksys 16 port GB switch. Linux clients are used to run http_load [24] stress tool. Each http_load stress tool can run up to 1000 concurrent requests per second. Multiple clients are used to measure the maximum load. For fair comparison, unnecessary services, processes and programs that add overhead to the OS-based client systems are disabled.

4.4.2 Measurements and Analysis

Figure 3 shows the plot for the number of requests per second versus the connection and response times. These measurements are made for a client file size of 3593 bytes. Note that these

times are very similar and follow the same pattern for variable requests. The bare PC Web server saturates at 6000 requests per sec, which indicates the maximum capacity of the server.

Figure 4 shows a graph for the variation in circular list queue size with respect to number of client requests. It is expected that as the number of requests increases, the maximum queue size increases which in turn increases the response time as shown in Figure 3. When more requests arrive in the RCV task gets busy thus making the response time slower. Currently, we only process one request for each invocation of the RCV task; if we modify the RCV task to process multiple requests, which already arrived then our response time will increase to some extent. This approach can be studied in further optimization. Notice that the queue size has much less effect on the connection time as it is the total processing time of a request and once a request is started it will complete sending data in a single thread of execution. Also, when an ACK comes from a client during the data transfer period, this triggers a resume operation thus taking the task from the queue and immediately processing it.

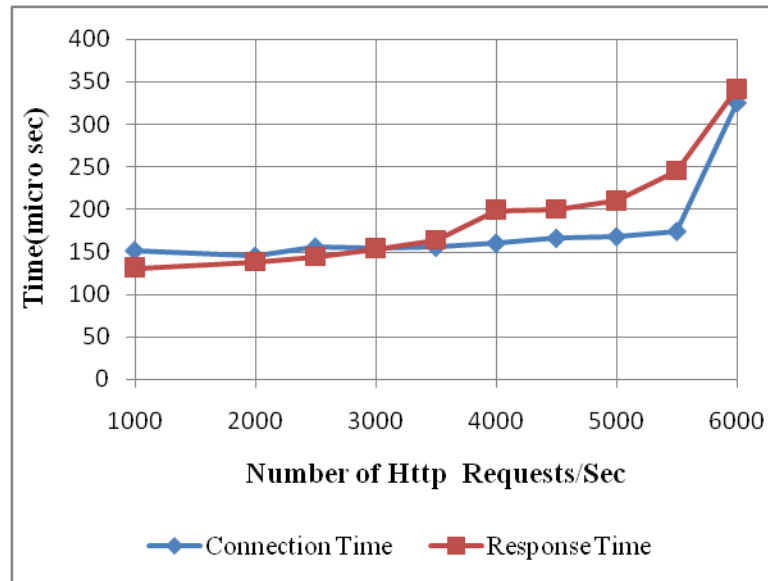


Figure 3. No. of Connections versus Processing Time

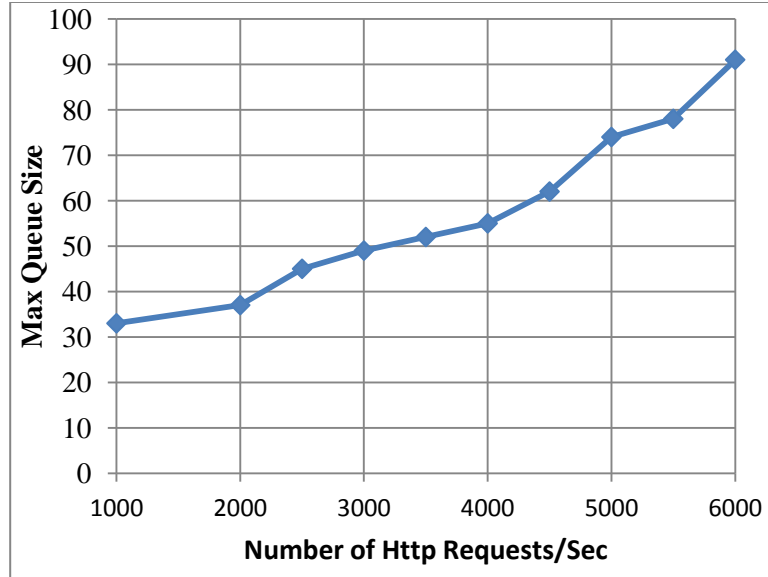


Figure 4. No. of Connections vs Max Queue Size

The Figure 5 shows the task utilizations with respect variation of requests for a file size of 3593 bytes. There are three types of tasks in the bare PC Web server: RCV, HTTP, and Main task. The Main task invokes RCV and HTTP tasks and there is very little processing involved in this task. As the graph indicates, the RCV task does more work than the HTTP task for this given file size. The Main task time shown in the chart indicates that as the number of requests increases, the Main task time reduced, which is an idle time in the Main task. When the idle time reaches close to zero, then the server saturates in its performance. Some of the idle time is also used to do some work in the Main task itself, which is negligible. For 6100 client requests, the CPU utilization for RCV, HTTP and Main task are 55%, 36%, and 9%.

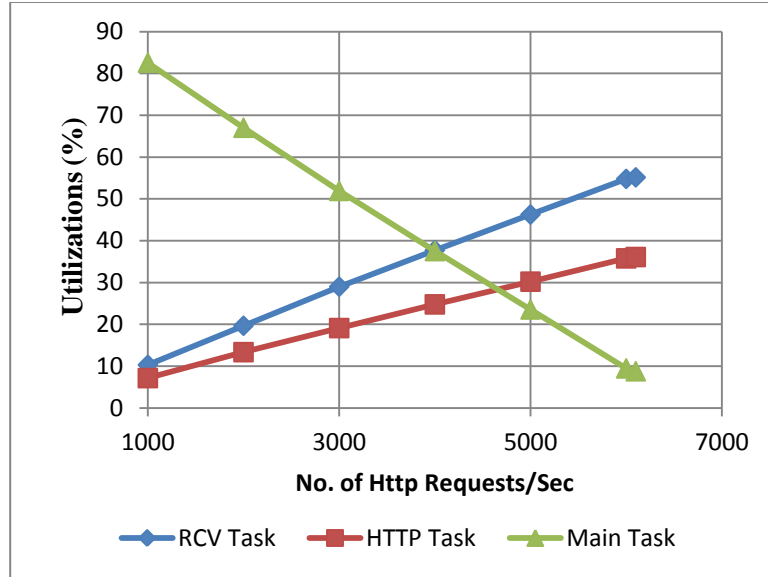


Figure 5. No. of Connections versus Task Utilizations

Figure 6 shows an interesting plot for HTTP and RCV tasks CPU utilizations. Notice that this chart indicates an almost a linear relationship of HTTP task with respect to a RCV task CPU utilization. This is quite unique to the bare PC Web server as there are only two dominant tasks running in the system. When more packets are received by the RCV tasks then there are more GET requests for the HTTP task. As the system is an application centric and runs only the intended functions, it is quite possible to predict the capacity of the bare PC Web server by using the above linearity behavior of RCV and HTTP tasks. Our future research can focus on studying such analytical model and compare it with the empirical results.

The insight into bare PC Web Server indicates that it is possible to design close to optimal systems when bare machine computing principles are used. It is also evident that the bare machine computing techniques can be used for any other servers or any other computer applications to design them for optimization. The performance measurements presented here in addition to the previous work in bare PC Web servers indicate that the performance predictions are easier to make in this type of systems.

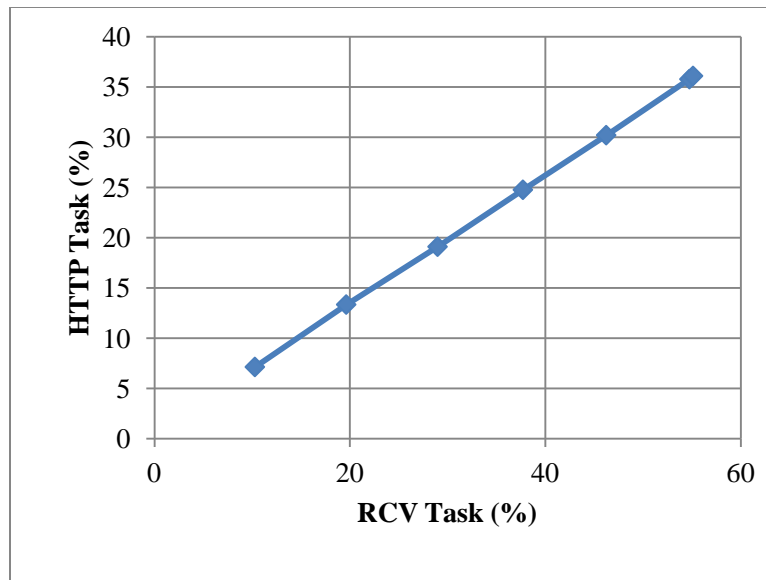


Figure 6. HTTP/RCV Task Utilization Plot

The linear behavior of RCV and HTTP tasks also illustrates that the bare PC Web server design is close to optimal and utilizes CPU very efficiently. This study can also serve as a close to optimal model for other server designs or other bare PC applications including split server applications.

5. SPLIT SERVERS

The novel splitting concept presented in this research 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 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 research demonstrate that 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 split architecture section.

The following sub-sections describe all initial work done in split servers including architecture, design and measurements.

5.1 *Split Protocol Architecture*

The split architecture used for the experiments described in this research is illustrated in Figure 7. Although these experiments were conducted in a LAN environment, 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) or S1 handles the {CE, CT} phases of a connection, and its delegated server S2 (DS) handles the {DT} phase. Similarly, S2 can act as a connection 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 $n (\geq 2)$ servers that can delegate requests to each other.

A given request is split at the GET command as shown in Figure 7. 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.

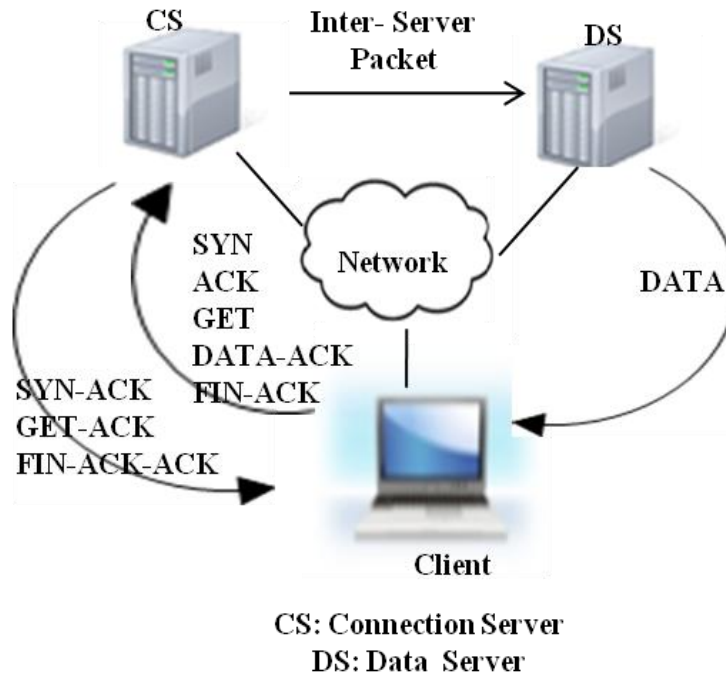


Figure 7. Split Architecture

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.

5.2 Design and Implementation

The detailed design and implementation of static and dynamic Web servers that run on a bare PC is described in [23]. 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 [35] 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 section 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 task. 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 research.

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 [40]. The state transition diagram (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.

5.3 Performance Measurements

5.3.1 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 [24] 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).

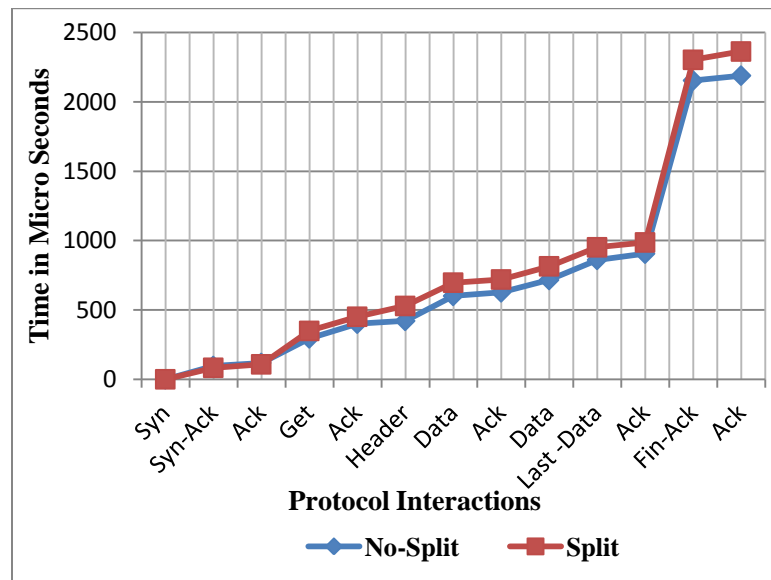


Figure 8. Internal Timings for HTTP/TCP

5.3.2 Measurements and Analysis

5.3.2.1 Internal Timing Comparison

Figure 8 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.

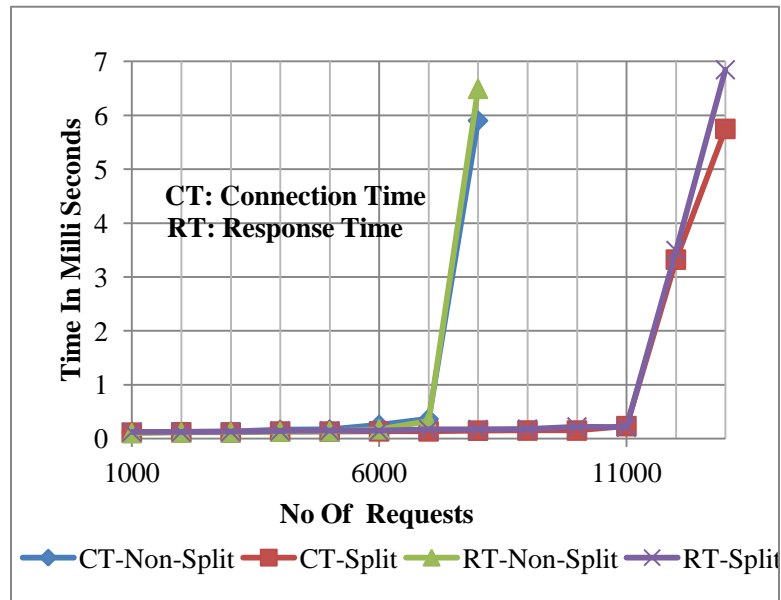


Figure 9. Response/Connection Times

5.3.2.2 *Response/Connection Times*

Figure 9 shows the response time and connection time at the client for varying HTTP request rates. S1 splits 50% of requests (with 0.5 probabilities) 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.

5.3.2.3 *CPU Utilization*

Figure 10 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.

5.3.2.4 *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 11. 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 12. 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 ($1000/12000$).

5.3.2.5 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 9. 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 13 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 9, 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).

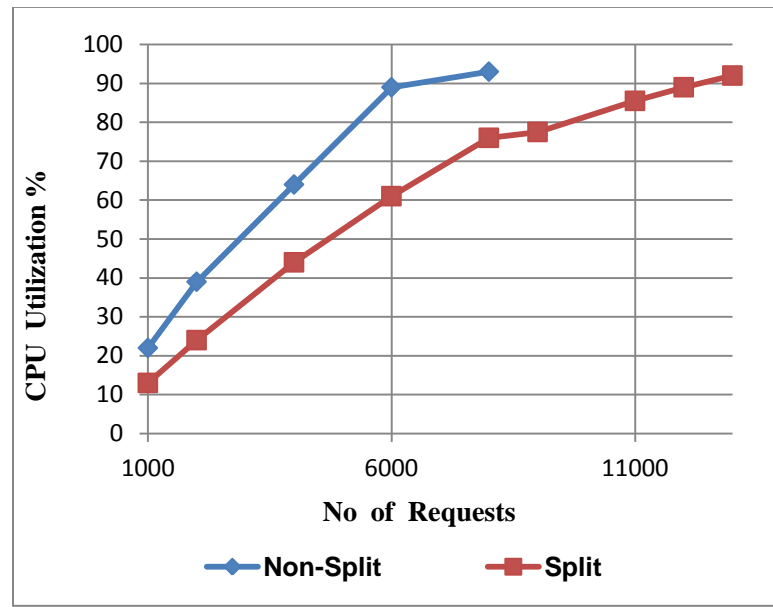


Figure 10. CPU Utilization

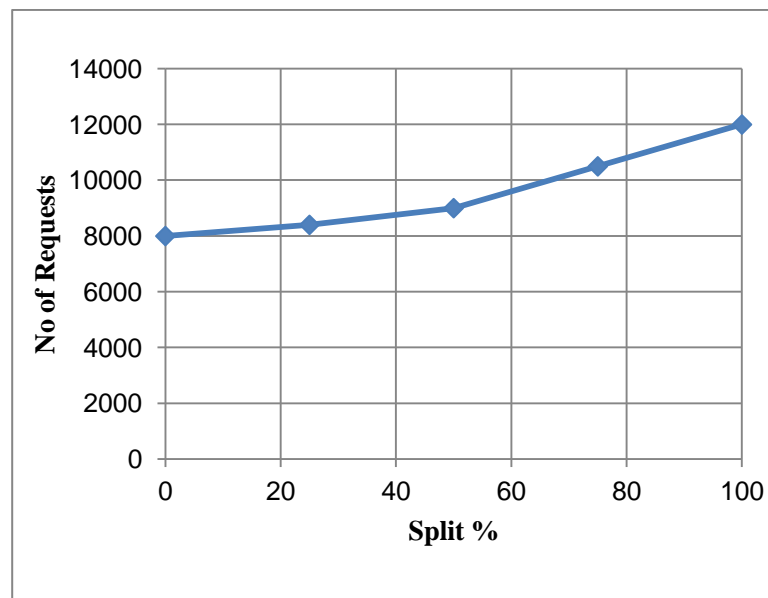


Figure 11. Split % (S1 Delegates to S2)

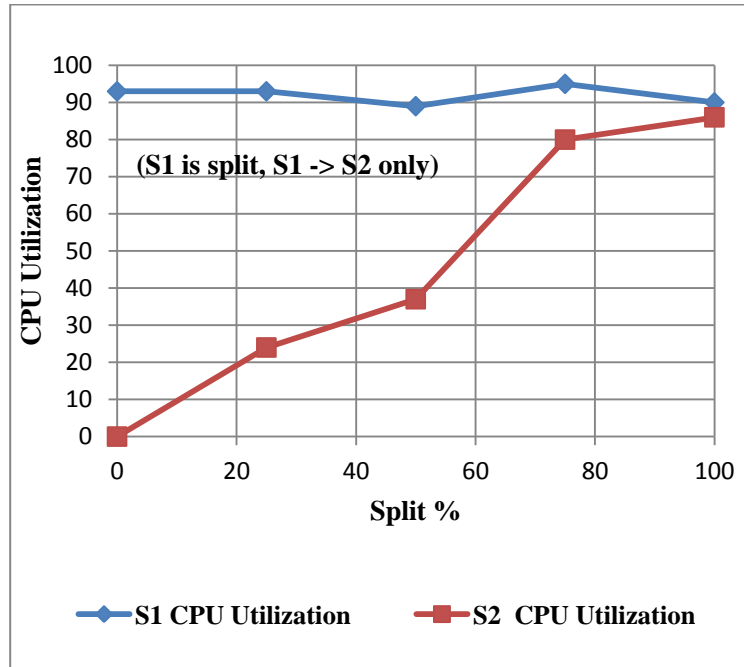


Figure 12. Split Server Utilization

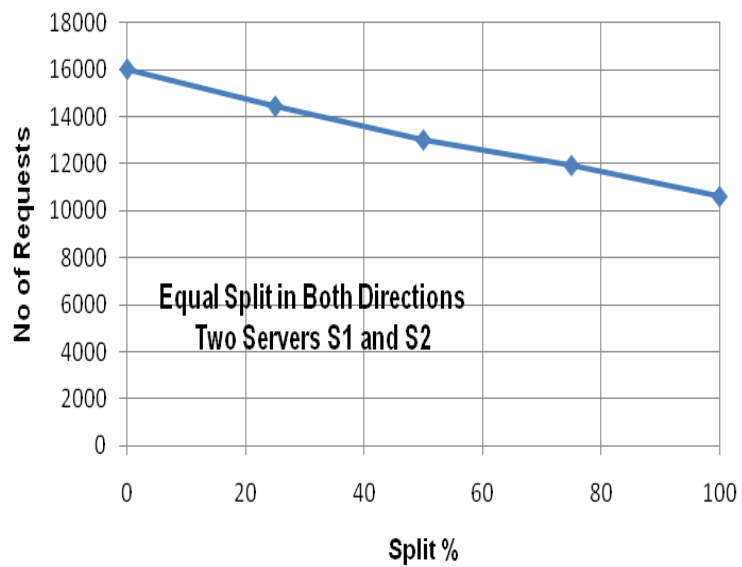


Figure 13. Equal Split in Servers

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 14 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.

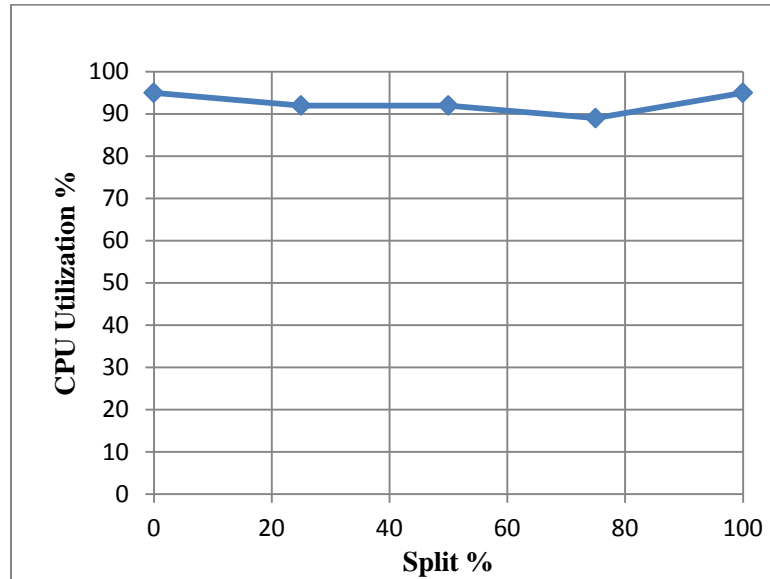


Figure 14. Varying Split Ratio on Both Servers

5.3.2.6 Varying Resource File Size

We varied the resource file size from 4K bytes to 32K bytes to determine its effect on splitting. Figure 15 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 9 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 16 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).

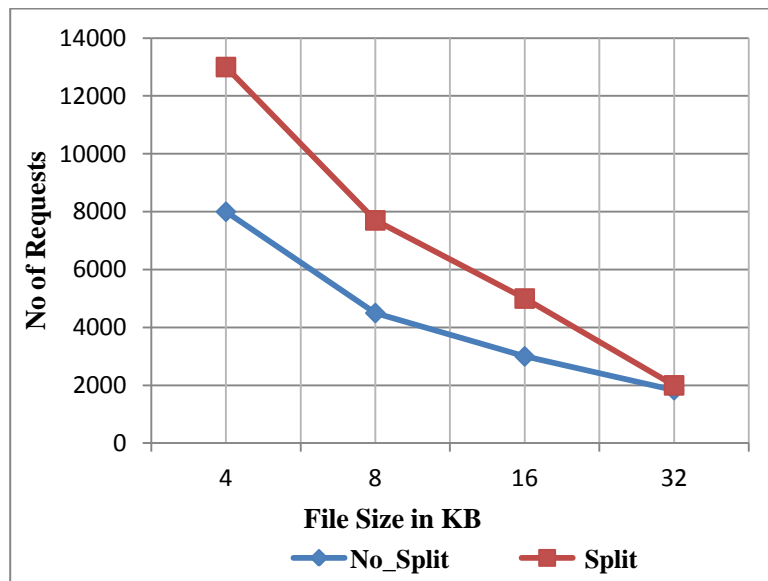


Figure 15. File Size Variations

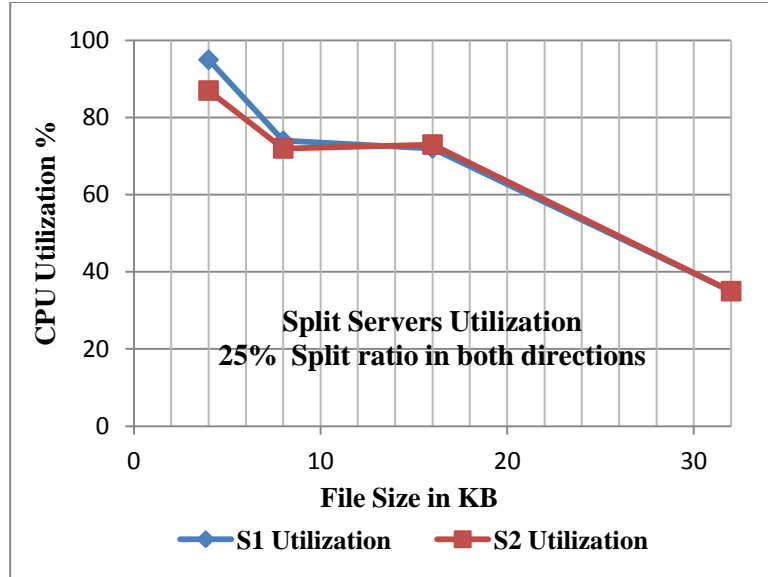


Figure 16. CPU Utilization 25% Split

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.

The experimental results also 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 section have potential applications in distributed computing and improving server reliability.

6. MINI-CLUSTER CONFIGURATION STUDY

The initial studies conducted on split servers as shown in section 5 helped us to conceive mini-cluster configurations as described in this section. The following sections describe cluster configurations, architecture, design and implementation and performance measurements conducted on mini-cluster configurations. Also varieties of protocols are proposed for reliable multicast with changing topology for a multi-hop mobile radio network [13]. Mini-Cluster configuration also offers better reliability because of interchangeable nature of CSs & DSs.

6.1 *Cluster Configurations*

Figure 7 illustrates generic request splitting [40] and shows the messages exchanged by the intertwined HTTP and TCP protocols. Connection establishment and termination are performed by one or more connection servers (CSs), and data transfer is done by one more data servers (DSs). When a request is split, the client sends the request to a CS, the CS sends an inter-server packet to a DS, and the DS sends the data packets to the client. Inter-server packets may also be sent during the data transfer phase to update the DS if retransmissions are needed. With partial delegation, the CS delegates a fraction of its requests to DSs. With full delegation, the CS delegates all its requests to DSs.

We consider mini Web server clusters consisting of two or more servers with protocol splitting. We then study cluster performance by measuring the throughput and connection and response times of three different server configurations with a varying number of CSs and DSs.

Configuration 1 in Figure 17 shows full delegation with one CS, one DS, and a set of clients sending requests to the CS. The DS and CS have different IP addresses, but the DS sends data to a client using the IP address of the CS.

Configuration 2 in Figure 18 shows a single CS with two or more DSs in the system with partial or full delegation. In partial delegation mode, clients designated as non-split request clients

(NSRCs) send requests to the CS, and these requests are processed completely by the CS as usual. The connections between the NSRCs and the CSs are shown as dotted lines. With full delegation, clients designated as split-request clients (SRCs) make requests to the CS, and these requests are delegated to DSs. For full delegation, there are no NSRCs in the system. When requests are delegated to DSs, we assume that they are equally distributed among the DSs in round-robin fashion. It is also possible to employ other distribution strategies.

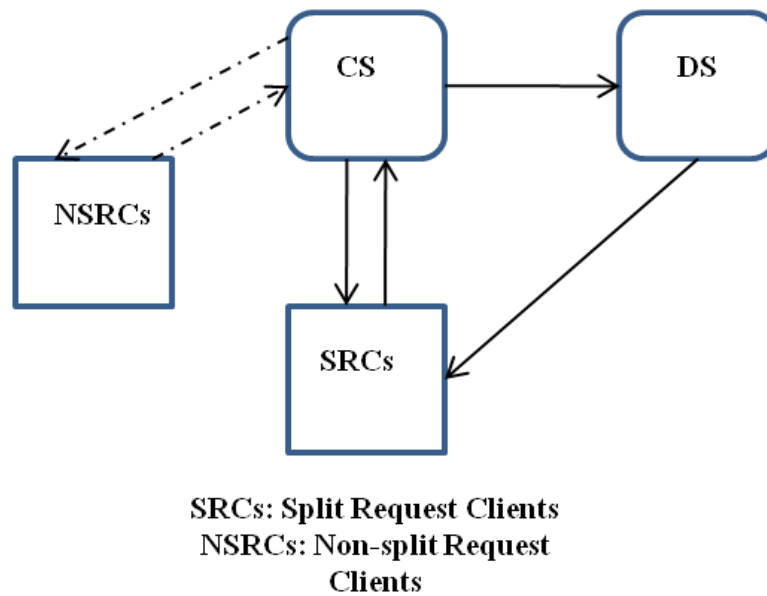
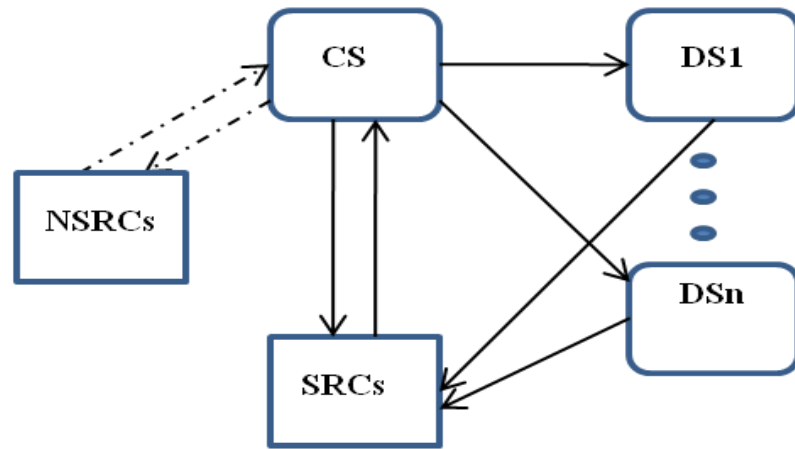


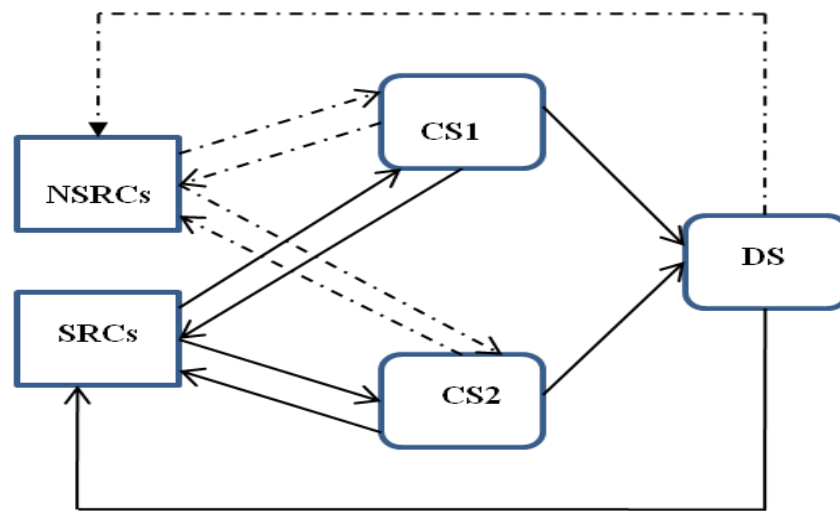
Figure 17. Split architecture configuration 1

Configuration 3 in Figure 19 shows two CSs and one DS with both SRCs and NSRCs. For this configuration, we used small file sizes to avoid overloading the single DS. Although we have not done so, multiple DSs could be added as in Configuration 3.



SRCs: Split Request Clients
NSRCs: Non-split Request Clients

Figure 18. Split architecture configuration 2



SRCs: Split Request Clients
NSRCs: Non-split Request Clients

Figure 19. Split architecture configuration 3

6.2 Performance Measurements

6.2.1 Configuration 1 (1 CS, 1 DS, full delegation)

In Figure 15, the performance of HTTP splitting with Configuration 1 was evaluated using various file sizes up to 32 KB. Here, we study the performance of Configuration 1 by varying the file size up to 128 KB and measuring the throughput in requests/sec. Figure 20 shows the results of these experiments. It can be seen that the performance of this configuration is worse than that of a two server non-split system for all file sizes. This is because the DS is overloaded resulting in performance degradation. However, the CS is underutilized since it is only handling connection establishment and termination. For a two server non-split system, we show the theoretical maximum performance (throughput) as being double that of a single (non-split) system, which was determined experimentally to be 6000 requests/sec. In practice, this theoretical limit for non-split systems will not be attained due to the overhead of load balancers and dispatchers.

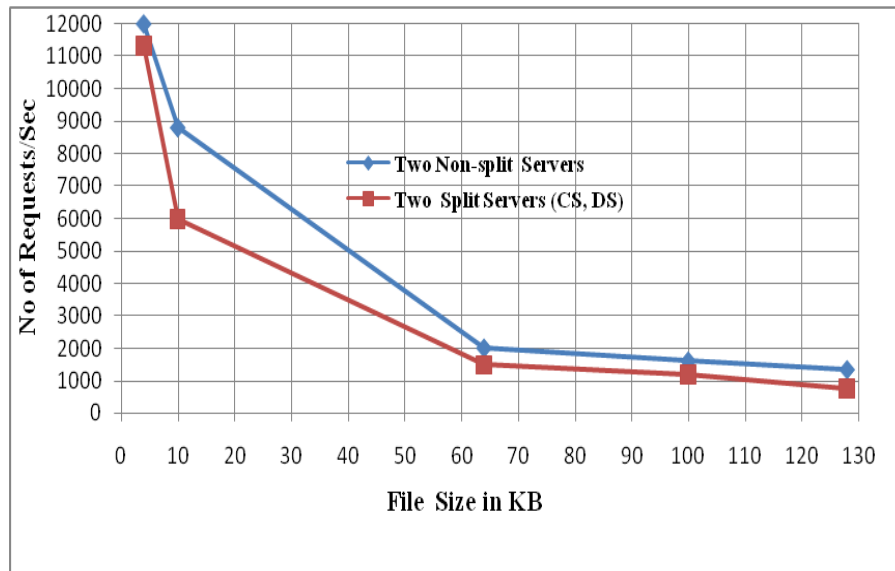


Figure 20. Throughput with increasing file sizes (Configuration 1)

Figure 21 shows the CPU utilization for the CS and DS in Configuration 1. The DS's CPU utilization for 64 KB files is close to the maximum, indicating that this configuration cannot

handle more than 1500 requests/sec. To get further insight into the performance limitations in this case, we determined the impact of connection and response time at the client due to increasing the request rate. The results are shown in Figure 22. The response time degrades as the number of requests increases starting at 1300 requests/sec and is largest at 1500 requests/sec as expected. These results suggest that performance may be improved by adding more DSs and utilizing the remaining capacity of the CS.

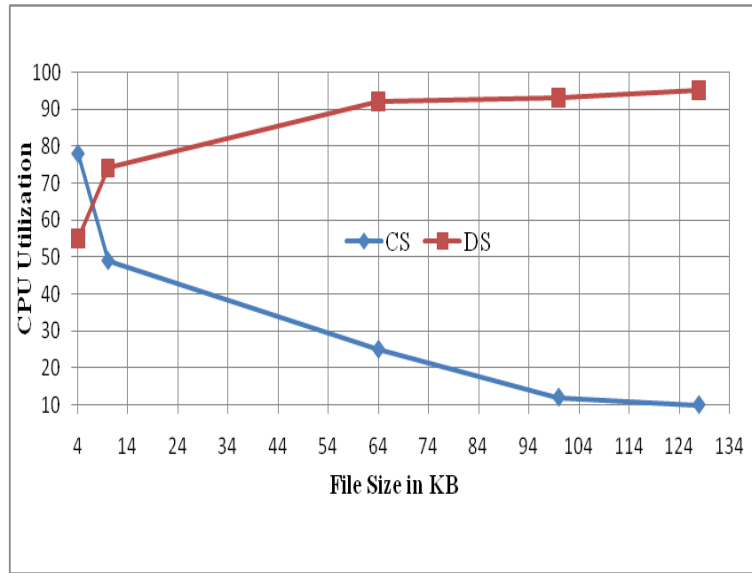


Figure 21. CPU utilization with increasing file sizes (Configuration 1)

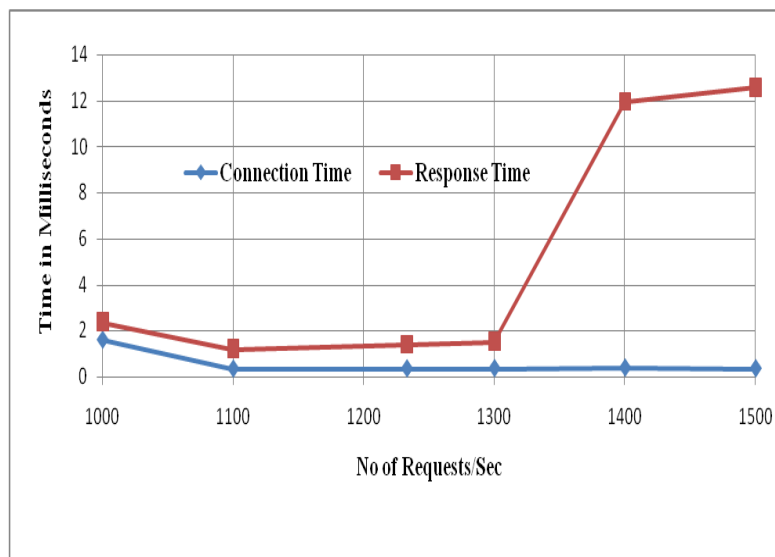


Figure 22. Connection and response times (Configuration 1, file size 64KB)

6.2.2 Configuration 2 (1 CS, 1-3 DS, full delegation)

Figure 23 shows the DS throughput for this configuration by varying the number of DSs with full delegation for 64 KB files. Adding more DSs improves the throughput as seen in the figure. With one DS (i.e. a two-server cluster), 1500 requests/sec can be handled versus the theoretical capacity of 2000 requests/sec for two non-split servers ignoring dispatcher or a load balancer overhead (about 75% of the theoretical non-split performance). With two DSs, the throughput increases to 2500 requests/sec (about 83.3% of the theoretical non-split performance). With three DSs, the maximum throughput is 3700 requests/sec compared to the theoretical limit of 4000 requests/sec for a non-split system (about 92.5% of the theoretical non-split performance).

Figure 24 shows connection and response times for Configuration 2 with 64 KB files. Although the response time for a single DS is poor, the average response and connection times improve significantly when the number of DSs is increased. A single (non-split) server has connection and response times of 1.62 ms and 2.38 ms respectively, compared to 365 μ s and 922 μ s respectively for a split system with three DSs and one CS (i.e., connection and response times are improved by factors of 4.4 and 2.6 respectively).

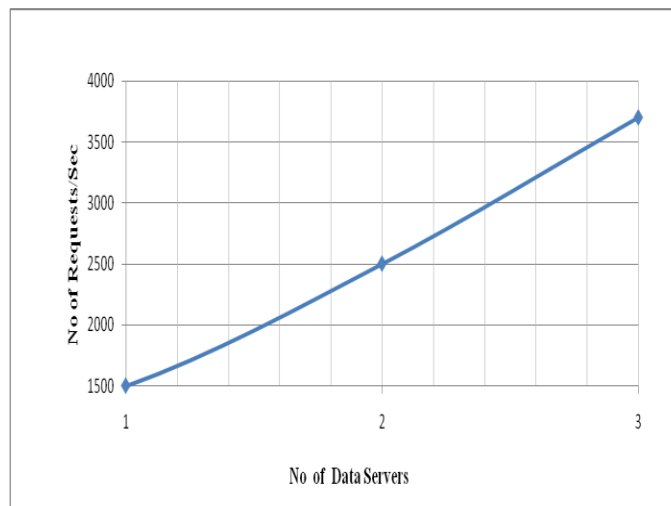


Figure 23. DS throughput (Configuration 2, file size 64KB)

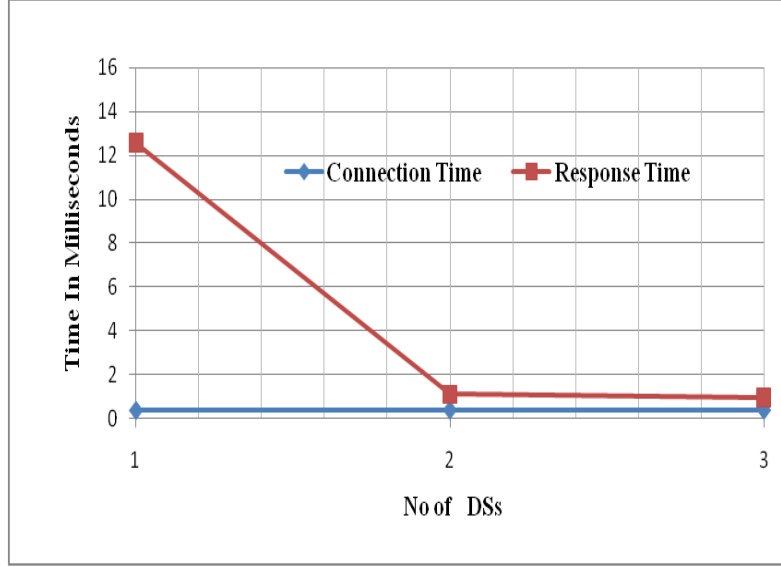


Figure 24. Connection and response times (Configuration 2, file size 64K)

Figure 25 shows the CPU utilization in Configuration 2 for the CS and DSs with 64 KB files. The DS utilization drops as expected due to load sharing, while the CS utilization increases due to the increased request rate. However, the CS still has unused capacity to support additional requests.

The preceding experiments show that the performance of a split system with a single CS and three DSs is close to the theoretical limit of a four-server non-split system with respect to both throughput, as well as connection and response times. In addition, the CS is still underutilized.

6.2.3 Configuration 2 (1 CS, 1-3 DS, partial delegation)

Configuration 2 with partial delegation and additional clients whose requests are not split (i.e., NSRCs) allows more load to be added in order to efficiently utilize the remaining capacity of the CS. The requests from NSRCs are completely processed by the CS, while the requests from SRCs are split. In this system, we have used 64KB files for requests.

Figure 26 compares the throughput for split servers with full and partial delegation. The throughput of the split system with partial delegation is more than the theoretical limit for a non-split system due to fully utilizing the capacity of the CS. For a split system with a single CS and a

single DS for 64 KB files, partial delegation improves the throughput by 25%. However, this performance gain does not scale up when more DSs are added since the CS is now close to capacity. For example, a split system with 3 DSs improves the throughput only by about 10%. These measurements indicate that a mini-cluster can only have a limited number of DSs if the system is to be self-contained (i.e., without using an external load balancer).

Figure 27 compares the throughput for split servers with full and partial delegation by varying the file size. The maximum throughput and a performance improvement of 25% are attained for 64 KB files with partial delegation. For 100 KB and 128 KB files, the performance improvements due to splitting are 17.7% and 12.5% respectively with partial delegation. Figure 28 shows connection and response times with partial delegation for 64 KB files. As with full delegation, response time with partial delegation is poor with only a single DS. However, the response time improves dramatically for split systems with two or three DSs and partial delegation. Figure 28 shows connection and response times with a partial delegation for 64 KB files. As with a full delegation, response time with the partial delegation is poor with only a single DS. However, the response time improves dramatically for split systems with two or three DSs and partial delegation.

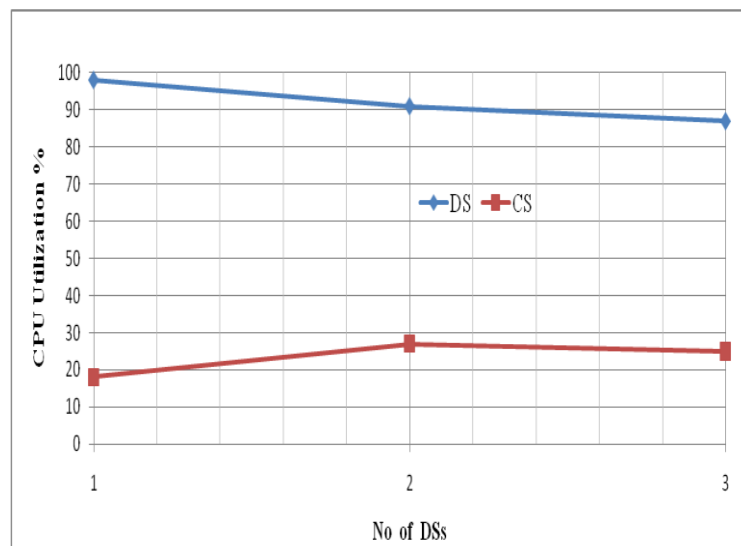


Figure 25. CPU Utilization (Configuration 2, file size 64KB)

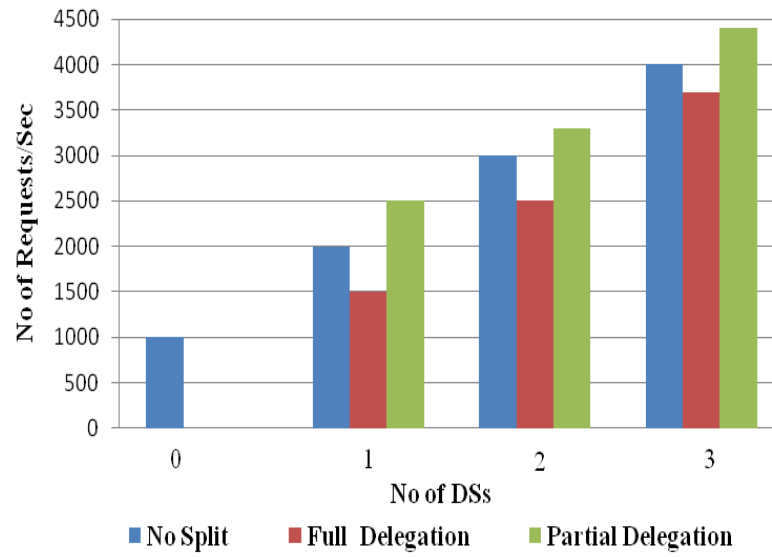


Figure 26. Throughput with full/partial delegation (Configuration 2, file size 64KB)

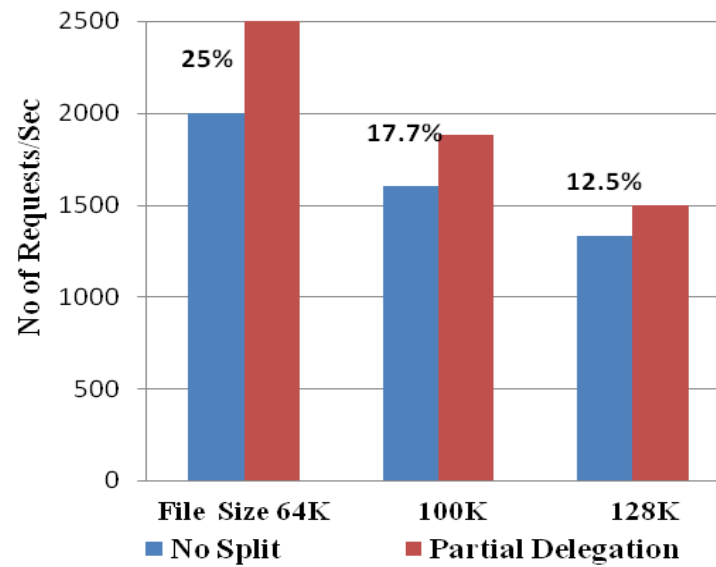


Figure 27. Throughput with full/partial delegation for varying file sizes (Configuration 2)

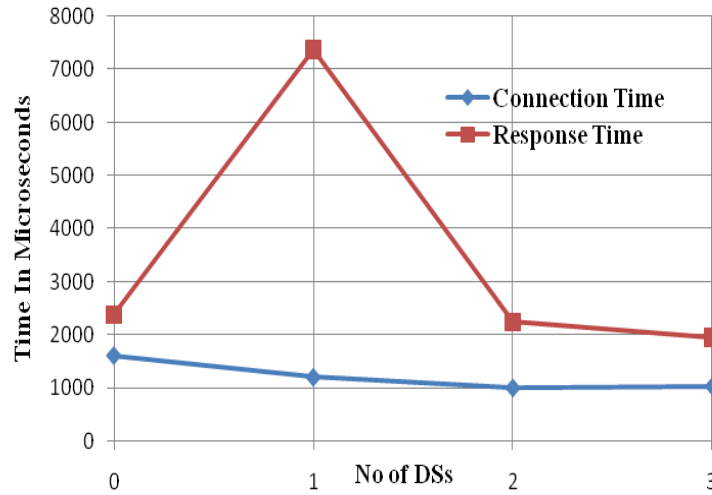


Figure 28. Connection and response times (Configuration 2, file size 64KB)

6.2.4 Configuration 3 (2 CS, 1 DS, partial delegation)

As before, requests are generated by a set of SRCs and NSRCs. For this configuration, 4 KB files were used since larger file sizes will overload the single DS. Figure 29 shows the throughput for three servers with full and partial delegation. Configuration 3 achieves a 6.5% throughput improvement over three non-split servers with full delegation; with partial delegation, it achieves a 22% improvement in throughput compared to three non-split servers.

Figure 30 shows the connection and response times for Configuration 3. As expected, response time is poor since the single DS gets saturated with the high request rate that be supported with two CSs. With partial delegation, response times improve significantly as the unused CS capacity is used to handle requests from the NSRCs without delegation. While the connection and response times using Configuration 3 are worse than for non-split servers, this disadvantage should be weighed against the increased throughput, cost, and security benefits of using a split system. Also, non-split servers will incur a reduction in response and connection times due to the overhead of using a dispatcher or load balancer.

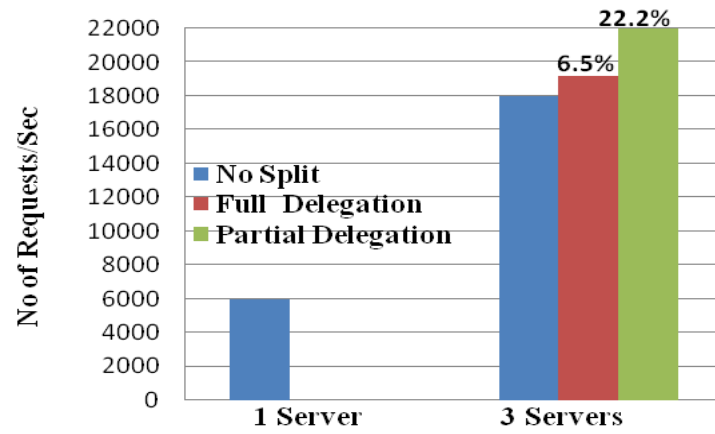


Figure 29. Throughput with full/partial delegation (Configuration 3, file size 4KB)

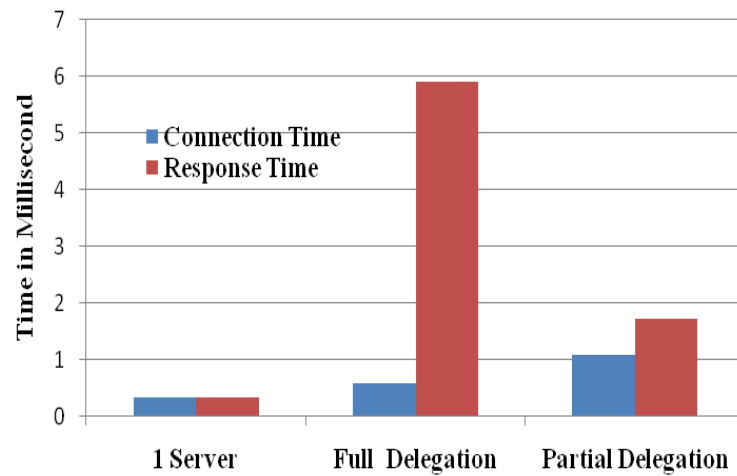


Figure 30. Connection times and response times with full/partial delegation

(Configuration 3, file size 4KB)

We studied the performance of mini Web server clusters with HTTP request splitting, which does not require a central load balancer or dispatcher, and is completely transparent to the client. Throughput as well as connection and response times with full and partial delegation of requests

were measured for a variety of file sizes. A split system with one CS and three DSs, and full or partial delegation, can be used to achieve response times, connection times and throughput close to, or better than, the theoretical limit of non-split systems. For example, this configuration with partial delegation achieves a 10% throughput increase for 64 KB files compared to four non-split servers and response times that are only slightly less. The same configuration with full delegation improves response times for 64 KB files by a factor of 2.6 over the equivalent non-split system, while achieving 92.5% of its theoretical throughput. For a split system with two CSs and one DS and partial delegation, a 22% improvement in throughput over three non-split servers is obtained for 4 KB files with response times that are close to those of a non-split system.

We also discussed the impacts of splitting. When evaluating the tradeoffs of splitting versus non-splitting, it is necessary to consider the overhead and cost of load balancers and dispatchers, which will result in less throughput and worse response times than the theoretical optimum values we have used for non-split systems. The experimental results appear to indicate that scalable Web server clusters can be built using one or more split server systems, each consisting of 3-4 servers. The performance of split servers depends on the requested file sizes, and it is beneficial to handle small file sizes at the CS and larger files with partial delegation to DSs. It would be useful to study performance of split server systems in which resource files of different sizes are allocated to different servers to optimize performance. More studies are also needed to evaluate the security benefits of split server clusters, and their scalability and performance with a variety of workloads. While these experiments used bare PC Web servers with no OS or kernel for ease of implementation, HTTP requests splitting can also be implemented in principle on conventional systems with an OS.

7. MODIFIED CLIENT/SERVER ARCHITECTURE

7.1 *Background*

In today's Internet world, client server architecture is very well-known in networking. It assumes a persistent connection between a client and a server. The design and implementation of clients and servers make this assumption to monitor their connections and provide appropriate responses. We modify this paradigm slightly to suit for split protocol concept. Our approach proposes this modification to accommodate split protocol implementation. The following sub-sections describe background, design and implementation, and experimental results.

An HTTP protocol intertwined with a TCP protocol is shown in Figure 1. The split protocol is studied in Section 5 and Section 6. In those cases, the CS handles all connection related to interfaces and communicates to the client in two directions. The DS only communicates to the client in one direction, i.e. sending data to the client. The CS also sends an inter-server packet to DS to provide client's request and its state. The CS is connected to the client throughout its session or during its processing of a given request. In such architecture one CS interfaces with one or more DSs to provide client services and thus becomes a bottleneck in a given mini-cluster configuration [41].

To address such bottleneck, we propose a split protocol at an architectural level thus resulting in a modified client server architecture, where connections and data transfers are separated entirely. In this approach, the data servers (one or more) can be located at a separate location than their counter part connection server. The CSs can be monitored for ongoing connections and the clients are isolated from data servers. The CS and DS servers can have a tight connection to serve client requests thus providing increased security at a server level. When a connection is established between a client and a CS, the CS will send an inter-server packet to a DS and terminate its connection processing, where a DS can finish the rest of the session to send

data and close the connection. Such modified client server interactions are shown in Figure 31. Notice that client sends interactions SYN, SYN-ACK-ACK and GET to CS and CS sends SYN-ACK and GET-ACK only to the client. After CS processes the connection, it sends a message to DS through an inter-server packet and eliminates this connection at CS. The rest of the connection and related interactions related to DATA, ACK and FIN-ACK will be dealt by DS. We have freed up CS completely after the GET is processed.

In real world applications, some servers may be close to data sources, and some servers may be close to clients. Splitting a protocol request and the underlying TCP connection in this manner allows servers to dynamically balance the workload. We have tested the splitting concept in a LAN that consists of multiple subnets connected by routers. In protocol splitting, clients can be located anywhere on the Internet. However, there are security and other issues that arise when deploying clusters in an Internet where a CS and a DS are on different networks [40].

Splitting protocol at a client server architecture level is different from migrating TCP connections, processes or Web sessions; splicing TCP connections; or masking failures in TCP-based servers. As per our knowledge, no work on splitting protocol connections at client server architectural level has been done before.

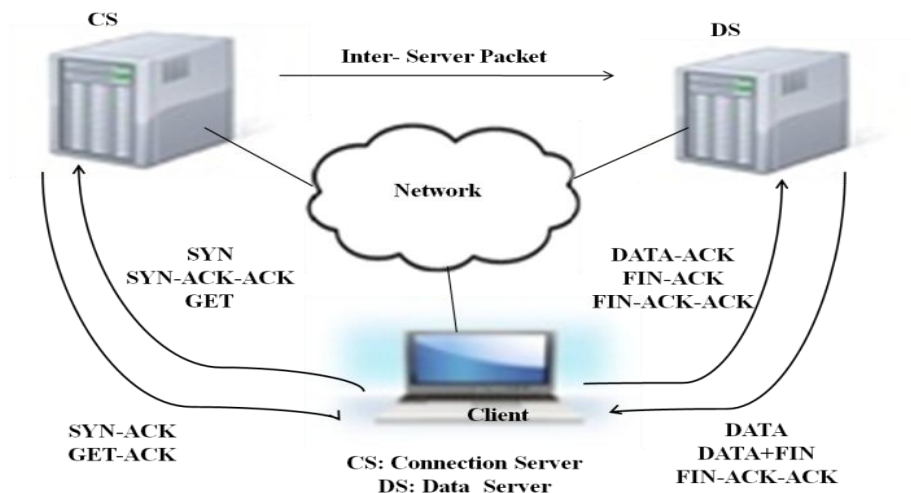


Figure31. Split Protocol Architecture

7.2 *Design and Implementation*

Split protocol client server architecture design and implementation differ from traditional client and server designs. As the traditional client server architecture is modified in this approach, we have designed and implanted a client and a server based on a bare PC, where there is no traditional OS or kernel running in the machine. This made our design simpler and easier to make modifications to conventional protocol implementations. Figure 32 shows a high level design structure of a client and server in a bare PC design. Each client and a server consist of a TCP state table (TCB), which consists of the state of each request. Each TCB entry is made unique by using a hash table with key values of IP address and a port number. The CS and DS TCB table entries are referred by IP3 and Port#. The Port# in each case is the port number of the request initiated by a client. Similarly, the TCB entry in the client is referenced by IP1 and Port#.

The TCB tables form the key system component in the client and server designs. A given entry in this table maintains complete state and data information for a given request. This entry requires about 160 bytes of relevant information and another 160 bytes of trace information that can be used for trace, error, log, and miscellaneous control. This entry information is independent of its computer and can be easily migrated to another PC to run at a remote location. This approach is not same as process migration [36] as there is no process information contained in the entry. The inter-server packet is based on this entry to be shipped to a DS when a GET message arrives from the client.

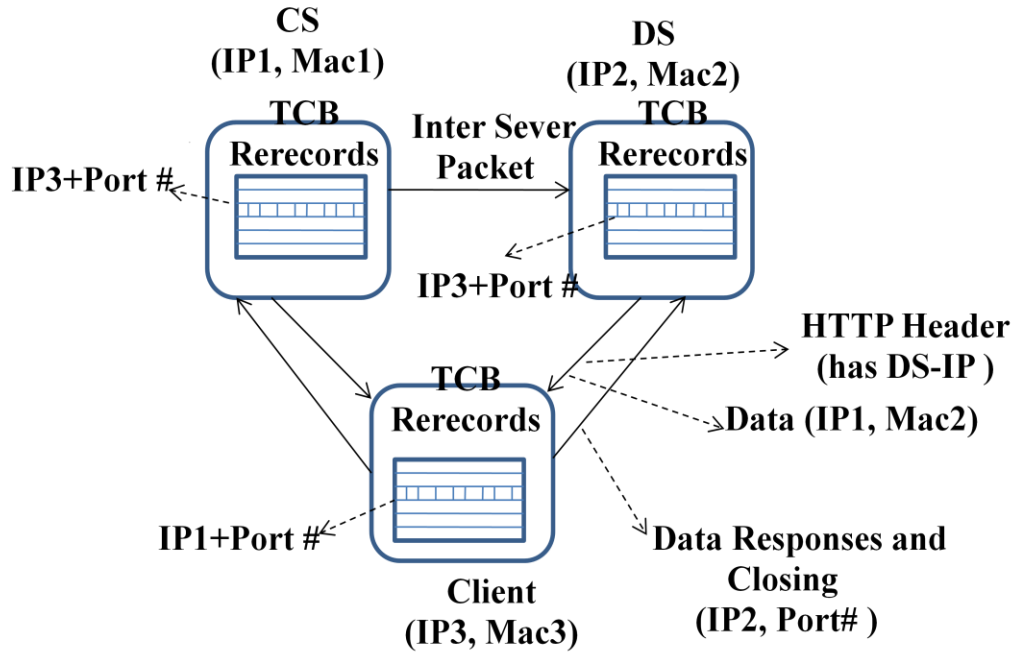


Figure 32. Design Structure

Notice that the client uses IP1 and Port# to address the TCB entry. That means, when DS sends data or other packets, then it must use IP1 as its source address and its own MAC address in the packet. However, a client must be aware of IP1 and IP2 addresses to communicate to two servers for different purposes. Client knows IP1 through its own request and by resolving the server's domain name. The client does not know IP2 address to communicate during the data transmission. We solved this problem by including the IP2 address in the HTTP header using a special field in the header format. In this design, a client could get data from any unknown DS and it can learn the data server's IP address from its first received data (i.e. header). This mechanism simplifies the design and implementation of split protocol client server architecture. This technique also allows the CS to distribute its load to DSs based on their CPU utilization without resorting to complex load balancing techniques [41].

We have taken an existing bare PC server design and created CS and DS elements. The CS design turned out to be fairly simple as its sliding window and data transmission logic is removed. The DS design also became somewhat simpler by removing the connection logic.

For a bare client implementation, a bare PC server design is modified by swapping the roles of client and server interactions. We had to create client request generator logic in addition to the server logic. The code and environment used for bare client and server are similar to split protocol servers described in Section 5 and Section 6.

7.3 *Experiment Results*

7.3.1 *Experimental Setup*

The experimental setup involved a prototype server cluster consisting of Dell Optiplex GX260 PCs with Intel Pentium 4, 2.8GHz Processors, 1GB RAM, and an Intel 1G NIC on the motherboard. All systems were connected to a Linksys 16 port 1 Gbps Ethernet switch. Bare PC clients were used to stress test the servers. The bare PC Web clients capable of generating 5700 requests/sec were used to create workload.

7.3.2 *Configurations*

Figure 33 shows a general configuration for connecting one CS, one or more DSs and one or more clients. All units are based on bare PC applications. Resource file size of 64K is used throughout our measurements. The CS will delegate all its requests to one or more DSs for data processing.

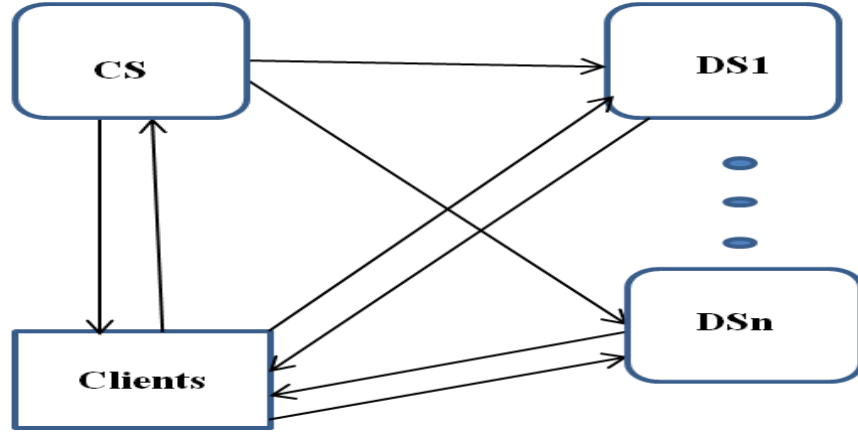


Figure 33. Split Architecture Configurations

7.3.3 Measurements: 1-4 DSs, Performance

Figure 34 shows our first set of measurements which are conducted using CS, 1-DS; CS, 2-DSs; CS, 3-DSs, and CS, 4-DSs. A minimum configuration of a split protocol client server system consists of one CS and one DS (a server pair). We measured that this pair can serve up to 900 Requests/sec for 64K file size. When the number of DSs is varied, then it shows a linear behavior in its performance improvement.

Figure 35 shows CPU utilizations for CS and DS. Notice that the CS utilization gradually increases up to 20% for 4 DSs. The DS utilization is maximized as we stress the server with peak capacity to conduct this experiment.

We expect the linear performance of the pair to continue until the CS gets saturated. The linear performance is also expected because the CS causes no bottleneck and all DSs execute concurrently and independently to process client requests. The number of servers connected to a single CS server can be estimated to be 15, by extrapolating from the above charts, the CS CPU time and the number of DSs. In similar study for split protocol based on conventional client server architecture, it is shown that one CS can support up to 4 DSs before it gets saturated [42]. The one CS and 4 DS configuration was referred to as a mini-cluster. This study based on split

protocol for the modified client server architecture indicates that the CS and DS based clusters can scale up to 15 DSs thus forming large cluster configuration for split protocols. Such clusters can be potentially used in building large server clusters similar to Google clusters [4].

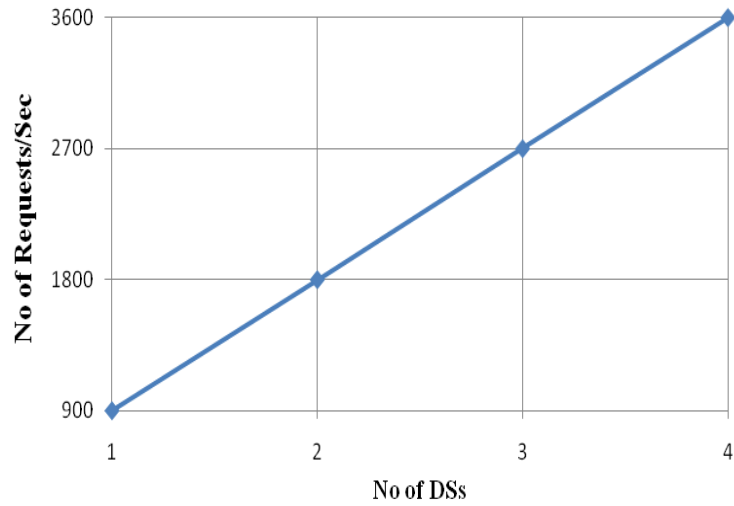


Figure 34. Varying DSs, Performance Chart

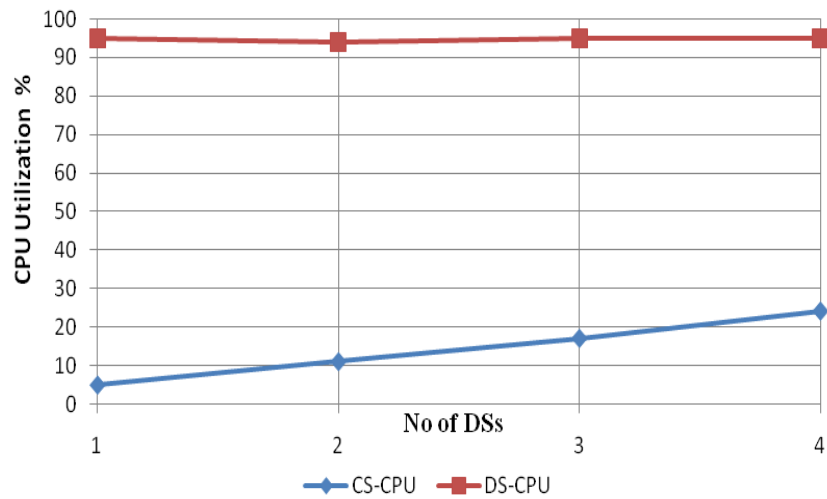


Figure 35. Varying DSs, CPU Utilization

7.3.4 Measurements: 1-4 DSs, Actual Times

Figure 36 shows the actual real time taken by a client server configuration for processing 1,620,000 total requests. Once again, these configurations indicate that the amount of time taken to process the total requests is inversely proportional to the number of DSs added to the system. Such behavior is very difficult if not impossible to realize in a typical cluster for servers. A typical cluster of servers also requires some external load balancing techniques and thus are prone to load balancing overhead. The proposed CS, DS cluster manages the load balancing internal to CS operation thus eliminating the load balancing overhead. In addition, the partitioning of the load for DSs at CS level is simpler as it has a tight communication with all its DSs. In our measurements, we used a round robin approach to delegate requests which costs no penalty in load distribution.

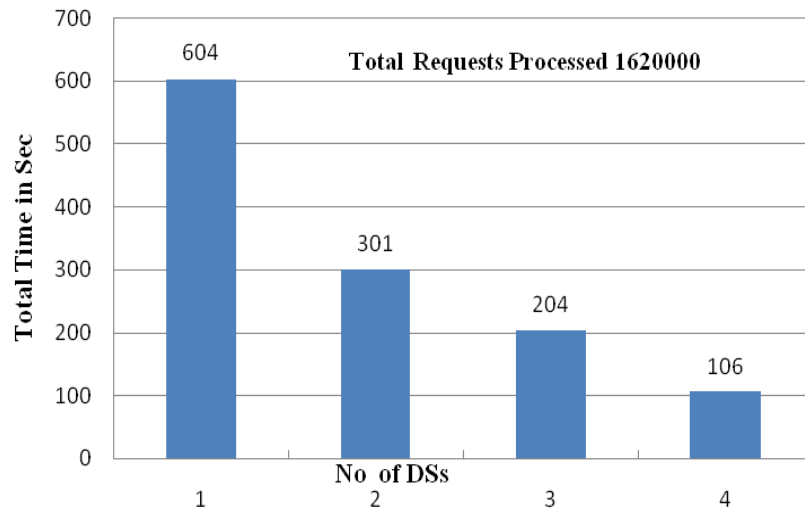


Figure 36. Varying DSs, Actual Times

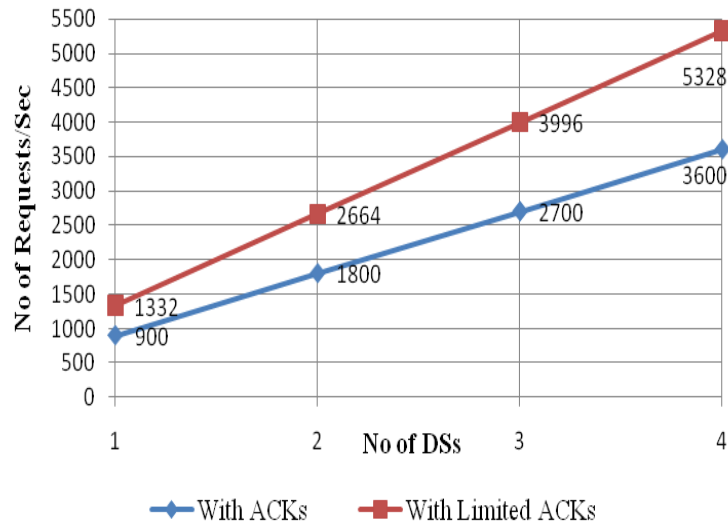


Figure 37. Varying ACKs, Performance Chart
Measurements: 1-4 DSs, Varying Acks

In order to improve DS performance further, we varied client responses (ACKs) for data by modifying the client code. Instead of sending ACK for each data packet received, we limited the ACKs to one for the entire data and one for closing the connection (FIN-ACK-ACK). This approach is similar to the concept of negative ACKs (ACK is only sent when data is not received). By reducing the data ACKs to a minimum, we measured the performance for one to four DSs as shown in Figure 37. This graph also plots the number of requests/sec for normal ACKs to make a relative comparison of data with ACKs and with limited ACKs. Observe that the limiting the ACKs for data, the performance improved by 48% at DS level.

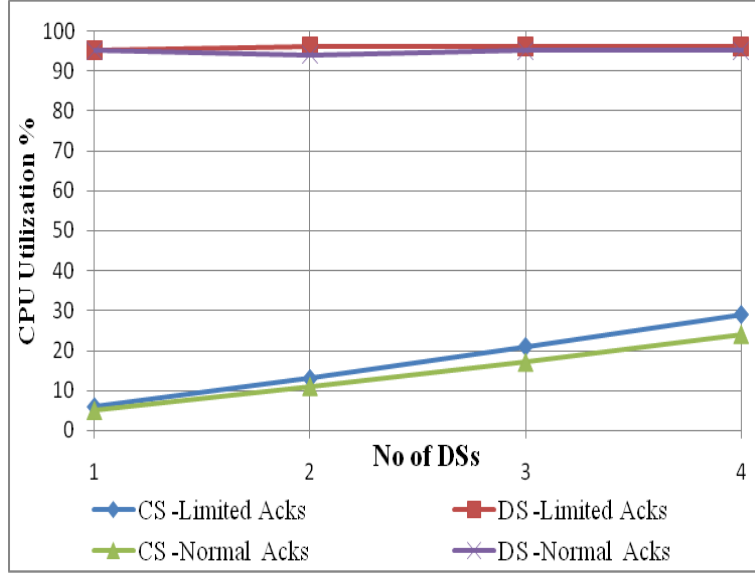


Figure 38. Varying ACKs, Utilizations

Figure 38 plot shows the CPU utilizations for CS and DSs with varying ACKs. The DS utilization has peaked due to maximum stress of the load from the clients. Surprisingly, the CS utilization increased with limited ACKs because now the CS is handling more requests than before.

With limited ACKs, the linear performance improvement continues up to 4 DSs. This is also expected as CS poses no bottleneck for 4 DSs. For limited ACKs, the number of DSs connected to a single CS can be estimated to be 13 by extrapolating the CS CPU time and the number of DSs.

Thus, a typical CS-DS cluster may contain up to 13 DSs with limited ACKs and 15 DSs with normal ACKs. As per Figure 37, variation of ACKs resulted in 48% improvement for 4 DSs. However, as per Figure 38, the CPU utilization increased from 24% to 29%, for the four DS system due to handling additional load. Thus, the 48% improvement shown for 4 DSs may not continue for large number of DSs in the system.

7.3.5 Lack of comparison with OS based systems

We could not provide any measurements for comparison of bare machine computing based servers and clients with respect to OS based systems, as it is very difficult to implement split protocols on them. Most of the OS based server and client systems do not provide easy access to split the protocol as we demonstrated in this section, and also in [41]. The bare machine computing and its application base allow easy access and modification to underlying protocols and their implementation. Our research findings provided in this section may provide motivation for other investigators to tackle the daunting task to implement split protocols on OS based systems.

7.4 Impacts of Modified Splitting Protocol

Splitting is a general approach that can be applied in principle to any application protocol that uses TCP (it can also be applied to protocols other than TCP to split the functionality of a protocol across machines or processors). In particular, splitting the protocol within a client server paradigm requires modification in the client server architecture. This approach impacts current server and client architectures and designs. However, this approach adds a new dimension and alternatives to current client server computing. Some of the issues and impact related to this novel approach are listed below:

- Split protocol configurations based on connections and data can be used for constructing large server clusters (4-15 DSs)
- A scalable performance can be achieved by adding DSs to the cluster without paying any penalty to load balancing overhead
- A uniform response time can be achieved by adding additional DSs as they work independently and concurrently in the system

- Complex load balancing techniques and dispatchers are not needed
- Connections and data transfers can be completely isolated in reference to clients (this may provide additional security due to data server isolation)
- Connection and data servers can be located in different places, especially data servers can be located in close proximity to data
- Client connections can be easily monitored without interrupting the client data communication
- Server designs can be simplified, especially the CS design is much simpler and manageable
- This approach can also be used for database servers and file servers.

The configurations studied and the results obtained in this section can be viewed as a first step to validate the applicability of splitting connections and data transfers as a general concept. In future, it would be of interest to investigate its applicability to other protocols and applications.

In this section we studied the performance of split protocols for connection and data servers for a modified client server computing system. This approach requires modifications to current client and server designs and implementations. We demonstrated scalable server architecture to construct large cluster of servers. We have shown some design and implementation details of constructing a bare PC based client server elements. We have shown performance improvements up to 48% in DS when a limited number of ACKs used for data transmissions.

We also discussed the impacts of splitting. When evaluating the tradeoffs of splitting versus non-splitting, it is necessary to consider the overhead and cost of load balancers and dispatchers,

which will result in less throughput and worse response times. We could not compare this approach with conventional servers and clients due to the difficulty in implementations. More studies are also needed to evaluate the security benefits of split server clusters, and their scalability and performance with a variety of workloads. While these experiments used bare PC Web servers and clients with no OS or kernel for ease of implementation, it may be possible implement these concepts in an OS based Web applications.

8. SIGNIFICANT CONTRIBUTIONS

Splitting is a general approach that can be applied in principle to any application protocol that uses TCP (it can also be applied to protocols other than TCP to split the functionality of a protocol across machines or processors). In particular, splitting the HTTP protocol has many impacts in the area of load balancing. We discuss some of these impacts below.

- Split protocol configurations can be used to achieve better response and connection times, while providing scalable performance. Splitting also eliminates the need for (and overhead/cost associated with) external load balancers such as a dispatcher or a special switch.
- Split protocol Configuration 2 (with one CS, one DS) and partial delegation achieves 25% more performance than two homogeneous servers working independently. This performance gain can be utilized to increase server capacity while reducing the number of servers needed in a cluster.
- Split server architectures could be used to distribute the load based on file sizes, proximity to file locations, or security considerations.
- The results obtained in this section (using specific machines and workloads) indicate that mini-cluster sizes are in the single digits. More research is needed to validate this hypothesis for other traffic loads. However, if we assume that mini-clusters should contain a very small number of nodes, they would be easier to maintain and manage (compared to larger clusters). Using mini-clusters, it is possible to build large clusters by simply increasing the number of mini-clusters.

- Splitting protocols is a new approach to designing server clusters for load balancing. We have demonstrated splitting and built mini-cluster configurations using bare PC servers. However, the general technique of splitting also applies to OS-based clusters provided additional OS overhead can be kept to a minimum (and that undue developer effort is not needed to tweak the kernel to implement splitting).
- When protocol splitting uses two servers (CS and DS) it dramatically simplifies the logic and code in each server (each server only handles part of the TCP and HTTP protocols unlike a conventional Web server that does both protocols completely). Thus, the servers are less complex and hence have inherently more reliability (i.e., are less likely to fail).
- Splitting can also be used to separate the “connection” and “data” parts of any protocol (for example, any connection-oriented protocol like TCP). In general, connection servers can simply perform connections and data servers can provide data. It can also be used to split the functionality of any application-layer protocol (or application) so that different parts of the processing needed by it are done on different machines or processors. Thus, a variety of servers or Web applications can be split in this manner. This approach will spawn new ways of doing computing on the Web.

In addition to the above list of items, we also consider some of the impacts of modified client server architecture innovations as listed in 7.4 are also our major contributions to the split protocol server and client arena. The configurations studied and the results obtained in this section can be viewed as a first step to validate the applicability of splitting as a general concept. In future, it would be of interest to investigate its applicability to other protocols and applications either on bare or OS based systems.

9. CONCLUSION

This doctoral dissertation introduced a novel concept called split protocol. The split protocol proposed here divides current single server architecture into a connection server (CS) and a data server (DS). The CS handles connection establishment and termination, and the DS handles data transfers. When a single server is divided into dual servers, it was discovered that it offers many unforeseen benefits. It was found that dual servers (CS, DS) outperform two single servers. In some cases, it outperformed about 25% more than two independent servers for the same workload. When dual servers are used, the CS has much less work to do and has more capacity left over. And the DS doing data transfers gets saturated at high workloads. Thus, a single CS can communicate with multiple DSs to constitute a mini-cluster configuration. The mini-cluster consists of one CS and 1-3 DSs providing better response time, connection time, and throughput than non-split systems.

In mini-cluster configurations, a CS can delegate all requests to DS to process data or it can also process some requests on its own resulting in a partial delegation. The variation of split can provide a new design option in server designs. When a CS performs partial delegation, we found that the mini-clusters always result in higher performance than conventional non-split servers.

The split protocol concept was further extended to modify current client server paradigm. If we make CS only provides connection establishment and DS provides data transfer and termination, then the CS has much more capacity to handle connection establishments. Also, there is no need for CS to wait for a request to be completely processed. However, this poses more workload on DS thus requiring more DSs in the cluster. This approach also results in a larger clusters consisting of one CS and many DSs. This technique requires the client to be aware of more than one DS, whom ever provides data for its connection. The client server architecture needs such modification to accommodate this new concept. We have demonstrated the idea by

using the HTTP header where we included the DSs address, so that the client now can communicate with this DS for ACKs and closing connections. This modification results in a greater benefit than one can expect. Now, there can be more than one DS providing the data for clients in an interleaved fashion.

The split concept demonstrated in this dissertation offers numerous benefits as mentioned before. As CS and DS consist of the entire state of a given request, it can be used as a fault-tolerant system. When a CS crashes, DS can function as CS and vice versa. There is no need to shadow servers if you use split server or mini-cluster configurations. There can be greater reliability of clusters achieved using split server configurations. The data can be placed on DSs and DSs can be located close to clients. The data locality can be used to configure clusters. The load balancing is done inherently without using any complex dispatching or load balancers in the clusters. The mini-cluster configurations are also easy to manage as they have small number of servers. The split protocol concept can also be used for other communicating systems and protocols. When split protocol servers and clients are used in the Internet environment, one can easily track user for connections without disrupting the data communication. The clients are completely shielded from DSs as they do not have direct communication with them during connection establishment. We also believe that this will provide more security to servers than a non-split protocol based servers. We have demonstrated split servers using the bare machine computing paradigm and run on bare PCs. The demonstrations were also limited to LAN environment as current WAN architecture is not amicable to the split architecture. However, the concept can be extended to OS based systems and WAN with some architectural modifications. When split protocol concept is widely used on the Internet, it could bring a major revolution in networking architectures and protocols and provide inherent security, load balancing, tracking, reliability and distributed clusters.

APPENDIX DATA TABLES

No of Requests / Seconds	Connection Time (Micro Seconds)	Response Time (Micro Seconds)
1000	151.1	130.5
2000	144.4	137.9
2500	155.2	143.8
3000	154.2	153.3
3500	154.9	163.4
4000	159.7	198.9
4500	165.5	199.3
5000	167.4	208.9
5500	173.7	244.6
6000	325.54	327.6

Table 1 for Figure 3: No. of Connections versus Processing Time

No of Requests / Seconds	Maximum Queue Size
1000	33
2000	37
2500	45
3000	49
3500	52
4000	55
4500	62
5000	74
5500	78
6000	91

Table 2 for Figure 4: Maximum Queue Size

No of Requests	RCV %	HTTP%	Main %
1000	10.303	7.1345	82.5625
2000	19.6492	13.3443	67.0065
3000	28.9999	19.0995	51.9006
4000	37.7314	24.7629	37.5057
5000	46.2538	30.1946	23.5516
6000	54.7714	35.7696	9.459
6100	55.1409	36.0885	8.7706

Table 3 for Figure 5: Connections vs. Max Queue Size

RCV %	HTTP%
10.303	7.1345
19.6492	13.3443
28.9999	19.0995
37.7314	24.7629
46.2538	30.1946
54.7714	35.7696
55.1409	36.0885

Table 4 for Figure 6: HTTP/RCV Task Utilization Plot

Processing Time	No-Split	Split
Syn	0	0
Syn-Ack	96	83
Ack	118	108
Get	295	349
Ack	401	451
Header	421	529
Data	601	697
Ack	627	720
Data	716	815
Last -Data	861	952
Ack	906	987
Fin-Ack	2155	2304
Ack	2190	2364

Table 5 for Figure 8: Internal Timings for HTTP/TCP

No of Requests	CT-Non-Split	CT-Split	RT-Non-Split	RT-Split
1000	0.125716	0.113994	0.105708	0.12407
2000	0.127928	0.120679	0.119219	0.12541
3000	0.134485	0.118946	0.124039	0.1309
4000	0.165464	0.134295	0.133748	0.14495
5000	0.171535	0.13335	0.13789	0.15034
6000	0.257208	0.138479	0.176372	0.14768
7000	0.368291	0.138479	0.32637	0.17445
8000	5.90293	0.151729	6.49287	0.17445
9000		0.153822		0.18135
10000		0.153822		0.22161
11000		0.235445		0.21355
12000		3.32499		3.49526
13000		5.7493675		6.84714

Table 6 for Figure 9: Response/Connection Times

No of Requests	Non-Split CPU %	Split CPU %
1000	22	13
2000	39	24
4000	64	44
6000	89	61
8000	93	76
9000		77.5
11000		85.5
12000		89
13000		92

Table 7 for Figure 10: CPU Utilization

Split%	No of Requests
0	8000
25	8400
50	9000
75	10500
100	12000

Table 8 for Figure 11: Split % (S1 Delegates to S2)

Split%	S1 CPU Utilization	S2 CPU Utilization
0	93	0
25	93	24
50	89	37
75	95	80
100	90	86

Table 9 for Figure 12: Split Server Utilization

Split Ratio	No of Requests
0	16000
25	14428
50	13000
75	11918
100	10600

Table 10 for Figure 13: Equal Split in Server

Split %	CPU Utilization
0	95
25	92
50	92
75	89
100	95

Table 11 for Figure 14: Varying Split Ratio on Both Servers

File Size (KB)	No_Split	Split
4	8000	14428
8	4500	7400
16	3000	5000
32	1833	2000

Table 12 for Figure 15: File Size Variations

File Size in KB	S1 Utilization	S2 Utilization
4	95	87
8	74	72
16	72	73
32	35	35

Table 13 for Figure 16: CPU Utilization 25% Split

File Size (kb)	Two Non-Split Servers	Two Split Servers
4	12000	11333
10	8800	6000
64	2000	1500
100	1600	1200
128	1333	766

Table 14 for Figure 20: Throughput with increasing file sizes (Configuration 1)

File Size in KB	CS	DS
4	78	55
10	49	74
64	25	92
100	12	93
128	10	95

Table 15 for Figure 21: Throughput with increasing file sizes (Configuration 1)

No of Requests	Connection Time	Response Time
1000	1.616961	2.38289
1100	0.338735	1.19293
1233	0.347395	1.40013
1300	0.35902	1.5338
1400	0.38095	11.9675
1500	0.35362755	12.59415

Table 16 for Figure 22: Connection and response times (Configuration 1, file size 64KB)

No of Servers	No of Requests
0	1000
1	1500
2	2500
3	3700

Table 17 for Figure 23: DS throughput (Configuration 2, file size 64KB)

No DSs	Connection Time	Response Time
1	0.35362	12.594
2	0.3702625	1.07056
3	0.36521	0.92209

Table 18 for Figure 24: Connection and response times (Configuration 2, file size 64K)

DSs	DS	CS
1	98	18
2	91	27
3	87	25

Table 19 for Figure 25: Connection and response times (Configuration 2, file size 64K)

No of DSs	No Split	Full Delegation	Partial Delegation
1	1000		
2	2000	1500	2500
3	3000	2500	3300
4	4000	3700	4400

Table 20 for Figure 26: Throughput with full/partial delegation
(Configuration 2, file size 64KB)

File Size	No Split	Partial Delegate
64	2000	2500
100	1600	1883
128	1333	1499

Table 21 for Figure 27: Throughput with full/partial delegation for varying file sizes
(Configuration 2)

DSs	Connection Time	Response Time
0	1616	2382
1	1208	7382
2	1008.33	2248.79
3	1041.02	1952.13

Table 22 for Figure 28: Connection and response times (Configuration 2, file size 64KB)

	1 Server	3 Servers
No Split	6000	18000
Full Delegation		19166
Partial Delegation		22000

Table 23 for Figure 29: Throughput with full/partial delegation
(Configuration 3, file size 4KB)

	Connection Time	Response Time
1 Server	346.4333333	340.4666667
Full Delegation	582.171	5916.86
Partial Delegation	1099.592	1731.14

Table 24 for Figure 30: Connection times and response times with full/partial delegation
(Configuration 3,4KB)

No of Ds	No of Requests/Sec
1	900
2	1800
3	2700
4	3600

Table 25 for Figure 34: Varying DSs, Performance Chart

No of Ds	CS-CPU	DS-CPU	No of Requests
1	5	95	900
2	11	94	1800
3	17	95	2700
4	24	95	3600

Table 26 for Figure 35: Varying DSs, CPU Utilization

No of DS	Total Time In Sec
1	604
2	301
3	204
4	106

Table 27 for Figure 36: Varying DSs, Actual Times

No DSs	No of Requests- A	No of Requests- B
1	900	1332
2	1800	2664
3	2700	3996
4	3600	5328

Table 28 for Figure 37: Varying ACKs, Performance Chart

No DSs	CS- Limited Acks	DS-Limited Acks	CS- Normal Acks	DS-Normal Acks
1	6	95	5	95
2	13	96	11	94
3	21	96	17	95
4	29	96	24	95

Table 29 for Figure 38: Varying ACKs, Utilizations

REFERENCES

- [1] A. Alexander, A. L. Wijesinha, and R. Karne. A Study of Bare PC SIP Server Performance, The Fifth International Conference on Systems and Networks Communications. ICSNC 2010, August 22-27, Nice, France.
- [2] A. Amis and R. Prakash, Load-balancing clusters in wireless ad hoc networks, in: *Proceedings of ASSET 2000*, Richardson, TX, March 2000, pp. 25–32.
- [3] G. Ammons, J. Appayoo, M. Butrico, D. Silva, D. Grove, K. Kawachiva, O. Krieger, B. Rosenberg, 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.
- [4] Barroso, L. A., Dean, J., and Urs Hölzle, U. 2003. Web search for a planet: The Google cluster architecture. *IEEE Micro* 23, 2, 22-28.
- [5] B. Ron C. Chiang, Anthony A. Maciejewski, Arnold L. Rosenberg, Howard Jay Siegel, "Statistical predictors of computing power in heterogeneous clusters," ipdpsw, pp.1-9, 2010 IEEE International Symposium on Parallel Distributed Processing, Workshops and PhD Forum, 2010
- [6] 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.
- [7] M. Chatterjee, S. K. Das and D. Turgut, An on-demand weighted clustering algorithm (WCA) for ad hoc networks, in: *Proceedings of IEEE GLOBECOM 2000*, San Francisco, November 2000, pp. 1697–1701.
- [8] B. Chen, Thesis: Multiprocessing with the Exokernel Operating System. Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, February 2000.
- [9] C. Coffing, An x86 Protected Mode Virtual Machine Monitor for the MIT Exokernel. Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, May 1999.
- [10] 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.
- [11] A. Emdadi, R. K. Karne, and A. L. Wijesinha. Implementing the TLS Protocol on a Bare PC, ICCRD2010, The 2nd International Conference on Computer Research and Development, Kuala Lumpur, Malaysia, May 2010.
- [12] Ciardo, G., A. Riska and E. Smirni. EquiLoad: A Load Balancing Policy for Clustered Web Servers". *Performance Evaluation*, 46(2-3):101-124, 2001.
- [13] A. Ephremides, J. E. Wieselthier and D. J. Baker, A design concept for reliable mobile radio networks with frequency hopping signaling, in: *Proceedings of IEEE*, Vol. 75(1) (1987) 56–73.

- [14] D. Engler, The Exokernel Operating System Architecture. Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, October 1998.
- [15] D. R. Engler and M.F. Kaashoek, "Exterminate all operating system abstractions," Fifth Workshop on Hot Topics in Operating Systems, USENIX, p. 78, Orcas Island, WA, May 1995.
- [16] B. Ford, M. Hibler, J. Lepreau, R. McGrath, and P. Tullman, "Interface and execution models in the Fluke Kernel," Proceedings of the Third Symposium on Operating Systems Design and Implementation, USENIX Technical Program, New Orleans, LA, February 1999, pp. 101-115.
- [17] B. Ford, and R. Cox, Vx32: "Lightweight User-level Sandboxing on the x86", USENIX Annual Technical Conference, USENIX, Boston, MA, June 2008.
- [18] G. Ford, R.K. Karne, A.L. Wijesinha, and P. Appiah-Kubi. "The design and implementation of a Bare PC email server," COMPSAC'09, 33rd IEEE International Computer and Applications Conference, pp. 480-485, July 2009.
- [19] G. Ford, R.K. Karne, A.L. Wijesinha, and P. Appiah-Kubi. "The performance of a Bare Machine email server," SBAC-PAD'09, 21st International Symposium on Computing Architecture and High Performance Computing, pp. 143-150, IEEE, October 2009.
- [20] Ford, G.H., Karne, R.K., Wijesinha, A.L., and Appiah-Kubi, P. The Performance of a Bare Machine Email Server, 21st International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD 2009), IEEE / ACM Publications, 28-31 October 2009, So Paulo, SP, Brazil, pp. 143-150
- [21] 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.
- [22] L. He, R. K. Karne, and A. L. Wijesinha, "Design and Performance of a bare PC Web Server," International Journal of Computer and Applications, vol. 15, pp. 100-112, Acta Press, June 2008.
- [23] L.He, Karne, R.K.Karne, A.L Wijesinha, and Emdadi. "A Study of Bare PC Web Server Performance for Workloads with Dynamic and Static Content,"HPCC 09", The 11th IEEE International Conference on High Performance Computing and Communications, pp. 494-499, Seoul, Korea, June 2009.
- [24] http://www.acme.com/software/http_load.
- [25] 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.
- [26] R. K. Karne, K. V. Jaganathan, and T. Ahmed, "How to run C++ Applications on a bare PC," SNPD'05, 6th ACIS International Conference, pp. 50-55, IEEE, May 2005.

- [27] R. K. Karne, "Application-oriented Object Architecture: A Revolutionary Approach," 6th International Conference, HPC Asia 2002, Centre for Development of Advanced Computing, Bangalore, Karnataka, India, December 2002.
- [28] R. K. Karne, K. V. Jaganathan, T. Ahmed, and N. Rosa. "DOSC: Dispersed Operating System Computing," OOPSLA '05, 20th Annual ACM Conference on Object Oriented Programming, Systems, Languages, and Applications, Onward Track, pp. 55-61, ACM, San Diego, CA, October 2005.
- [29] G. H. Khaksari, A. L. Wijesinha, R. K. Karne, L. He, and S. Girumala, "A Peer-to-Peer Bare PC VoIP Application," CCNC'07, Proceedings of the IEEE Consumer and Communications and networking conference, pp. 803-807, IEEE Press, Las Vegas, Nevada, January 2007.
- [30] H. Kim, V. Pai, and S. Rixner. "Increasing web server throughput with network interface data caching," 10th International conference on Architectural support for programming languages and operating systems, San Jose, California, October 2002.
- [31] <http://news.bbc.co.uk/2/hi/technology/5107642.stm>. 30 million PCs being dumped each year in the US alone." BBC News: PC users ...
- [32] S. Lampoudi and D.M. Beazley. "SWILL: A Simple Embedded Web Server Library," FREENIX Track: 2002 USENIX Annual Technical Conference, Monterey, California, June 2002.
- [33] 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.
- [34] A.B. McDonald and T.F. Znati, A mobility-based framework for adaptive clustering in wireless ad hoc networks, IEEE Journal on Selected Areas in Communications 17(8) (1999) 1466–1487
- [35] D.S. Milojevic, F. Douglass, Y. Paindaveine, R. Wheeler and S. Zhou. "in proxy-based session handoff ion," ACM Computing Surveys, Vol. 32, Issue 3, September 2000, p241-299
- [36] D.S. Milojevic, F. Douglass, Y. Paindaveine, R. Wheeler and S. Zhou. "Process Migration" ACM Computation Surveys Vol.32, Issue3, September 2000, pp-241-299
- [37] A.K. Parekh, selecting routers in ad-hoc wireless networks, in: *Proceedings of the SBT/IEEE International Telecommunications Symposium*, August 1994.204.
- [38] 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), pp. 37-66, ACM, February 2000.
- [39] B. Rawal, R. Karne, and A. L. Wijesinha. "Insight into a Bare PC Web Server," CAINE-2010, 23rd International Conference on Computer Applications in Industry and Engineering,

November 8-10, 2010, Imperial Palace Hotel, Las Vegas, Nevada USA.

- [40] B. Rawal, R. Karne, and A. L. Wijesinha. Splitting HTTP Requests on Two Servers, The Third International Conference on Communication Systems and Networks: COMPSNETS 2011, January 2011, Bangalore, India
- [41] B. Rawal, R. Karne, and A. L. Wijesinha. "Mini Web Server Clusters for HTTP Request Splitt", 13th International Conference on High performance Computing and Communication, HPCC-2011, Banff, Canada, I Sept 2-4,2011.
- [42] K. Sultan, D. Srinivasan, D. Iyer and L. Iftod. "Migratory TCP:Highly Available Internet Services using Connection Migration," Proceedings of the 22nd International Conference on Distributed Computing Systems, July 2002
- [43] "Tiny OS," Tiny OS Open Technology Alliance, University of California, Berkeley, CA, 2004, I <http://www.tinyos.net/>.
- [44] The OS Kit Project," School of Computing, University of Utah, Salt Lake, UT, June 2002, <http://www.cs.utah.edu/flux/oskit>.
- [45] T. Venton, M. Miller, R. Kalla, and A. Blanchard, "A Linux-based tool for hardware bring up, Linux development, and manufacturing," IBM Systems Journal, Vol. 44 (2), pp. 319-330, IBM, NY, 2005.
- [46] Sujit Vaidya and Kenneth J.Chritensen, "A Single System Image Server Cluster using Duplicated MAC and IP Addresses," Proceedings of the 26th Annual IEEE conference on Local Computer Network (LCN'01)
- [47] 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.
- [48] R. Yasinovskyy, A. L. Wijesinha, R. K Karne and G. Khaksari. "Comparison of VoIP Performance on IPv6 and IPv4 Networks", The 7th ACS/IEEE International Conference on Computer Systems and Applications I (AICCSA), 2009.
- [49] 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.

Bharat Rawal

740 Camberley Circle, Towson, MD 21204 USA
Cell 443-889-1308 email:brawal@towson.edu

Objectives

Seeking for an academic position where my teaching and research experience in addition to the intellectual potential can be utilized in a classroom or in a research environment at an academic institution.

Education

D.Sc.	Information Technology, Towson University,	Expected Dec 2011.
M.B.A.	Towson University,	December 2008.
M.Sc.	Physics, South Gujarat University, India,	May 1989.
B.Sc.	Physics, South Gujarat University, India,	May 1986.

Doctoral Thesis

Implementation of network and application protocols is usually done on a single server environment. However, there has been increasing interest to develop migratory protocols such as M-TCP which enable servers to delegate their requests to other servers due to heavy load conditions or server failures. Most of the migratory protocol approaches move the entire protocol session to another server without involving a client. This dissertation takes the migratory idea to a radical approach where the protocol itself splits among two servers. When a protocol such as HTTP combined with TCP is split in two ways; it results in one server providing the connection establishment and closing and the other server providing the data transfer. The communication between a client and its original connected server remains the same, and the session is totally transparent to the client. This approach has resulted in many interesting server configurations and advocates optimal mini-cluster configurations for designing high performance clusters. This thesis focus on studying and conducting performance measurements on various connection and data servers and developed optimal load balancing techniques based on Web server resource file sizes and full or partial delegation of client requests. The split protocol concept demonstrated in this thesis has broader implications in general on any protocol design and its implementation. It also allows users to dedicate servers which only supply data, and they are completely hidden from their clients. This approach may also suggest better techniques for achieving more security in servers. This work also

serves as a cornerstone for developing and implementing future network protocols based on a split concept.

Research Experience

Over the three year period, the doctoral thesis work has provided me with incredible experience and knowledge. During this time, I have learned how to identify and solve problems in real world. The split protocol thesis was implemented on a bare machine server, where there is no operating system or a kernel. Understanding and mastering the bare machine computing posed me with a daunting challenge. Building bare machine Web servers and clients based on the split concept provided me with vast knowledge and experience in software development and testing. It also enabled me to master various networks protocols including: HTTP, TCP/IP, and Ethernet. Numerous benchmark tools such as “httpref,” “http_load,” and “Wire-shark” were used during the research. Other network elements such as Gateways, Switches, and Routers were used to build a local Internet to conduct performance measurements. Installation and configuration of Linux expertise were needed to set up and run clients on Linux systems.

In addition to the above practical experience, I have mastered the academic aspect of research including: literature search, identifying research issues, solving technical problems, planning and executing research plans, writing peer reviewed papers and presenting at professional conferences.

Research Publications

- (1) B.Rawal, R. Karne, and A. L. Wijesinha. “Splitting HTTP Requests on Two Servers.” The Third International Conference on Communication Systems and Networks: COMPSNETS 2011, January 2011, Bangalor, India.
- (2) B. Rawal, R. Karne, and A. L. Wijesinha. “Insight into a Bare PC Web Server.” CAINE-2010, 23rd International Conference on Computer Applications in Industry and Engineering, November 8-10, 2010, Imperial Palace Hotel, Las Vegas, Nevada USA.
- (3) B. Rawal, R. Karne, and A. L. Wijesinha. “Mini Web Server Clusters based on HTTP Request Splitting” HPCC 2011 : The 13th IEEE International Conference on High Performance Computing and Communications ,Sep 2, 2011 - Sep 4, 2011, Banff, Canada
- (4) B. Rawal, R. Karne, and A. L. Wijesinha. “Split Protocol Client/ Server Architecture” Submitted to ISCA.

Teaching Experience

I have taught a variety of courses at an undergraduate level over past four years during the graduate study at Towson University. The teaching experience spreads across multiple disciplines, including: computer science, physics, and information management. This background clearly reflects my undergraduate and graduate education. Some of the courses include COSC109 (Computers and Creativity), PHY100 (Understanding

Physics), PHY212 (General Physics II – Electricity, Magnetism, & Light), MGMT 337 (Information Technology Management). In addition I have also acted as a substitute teacher for graduate courses at Towson University for COSC519 (Operating Systems-I) and COSC650 (Data Communication Networks).

Teaching Methodology

My teaching methodology focuses on various aspects of teaching and learning strategies. In the beginning of a class, I learn the student background and their depth of knowledge. Clear objectives of a course will be given to students in the beginning of class. In every class student will be given an opportunity to ask questions and make the class interactive and live. The teaching will involve state-of-the art tools and Web-based learning. Many hands-on experiments and challenging homework assignments and projects will be given to students. Students will be always encouraged to explore new ideas and participate in discussions and group learning. Each semester, I take student evaluations seriously; consider their comments and criticism, and take appropriate actions to correct my teaching style and methodology. I relentlessly improve my teaching knowledge and methodology by reading, involving in research, attending professional conferences, participate in professional organizations and working with other peers.

Industrial Experience

Over a ten year period, I have gained value added experience in a pharmaceutical company in India. Starting with a junior level marketing job, I was rapidly promoted to a high level manager in the organization, where I made significant contributions to the organization. This experience improved my personal, communication and managerial skills. During this period, the broad experience gained varied from marketing new products, branding products, dramatically improving sales, managing employees successfully, help employees to increase their productivity and job retention. This broad people's skills helped me to become an effective teacher and a researcher.

Professional Affiliations

Member of International Society for Computers and Their Applications (ISCA), IEEE, American Marketing Association (AMA), Honor society of Marketing Mu Kappa Gama and International Student Organization,

Service Activities

- Department seminars and presentations
- Marketing research for WYPR public radio, 2008
- Developed various educational tools (Inertia-less magnetometer)
- Student Advising
- Tutoring.

Honors and Awards

- World's coolest business writing competition "Moshpit," Grater Baltimore Technology Council, 2008
- International Student Scholarship award for Books: Merick Business school academic achievement scholarship awards 2008
- Merit Scholarship, South Gujarat University, India.

References

Dr. Ramesh K. Karne, Professor and Dissertation Chair
Department of Computer and Information Sciences
Towson University, Towson, MD 21252
rkarne@towson.edu
(410) 704-3955

Dr. Alexander L. Wijesinha, Professor
Department of Computer and Information Sciences
Towson University, Towson, MD 21252
awijesinha@towson.edu
(410)- 704-4660

Dr. Josh Dehlinger, Professor
Department of Computer and Information Sciences
Towson University, Towson, MD 21252
JDehlinger@towson.edu
(410)- 704-4536

