Split Protocol Client/Server Architecture

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

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

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

***Abstract*—Protocol splitting has been used to enable protocols to be split at a server level without client involvement. We describe a novel split protocol client/server architecture that completely separates connections and data transfers within a typical session. In this approach, a client becomes aware of its multiple server sources and communicates with them using their IP addresses. Specifically, a client makes a single TCP connection to a connection server and subsequently communicates with one or more data servers to obtain its data and close the connection. We also conduct experiments and measure performance to demonstrate the feasibility of this architecture. Our results indicate that scalable server cluster configurations can be built using this approach. The proposed architecture simplifies server implementations, avoids traditional load balancing techniques, and isolates clients from data servers. It also results in a scalable and distributable approach to client/server computing that provides an alternative to the current paradigm.**

Keywords-Split Protocol, Performance, Client Server Computing, Server Cluster, Web Servers, Bare Machine Computing.

I. INTRODUCTION

Network protocols are typically implemented as layered entities within an OS-controlled stack. Alternatively, they can be implemented in an intertwined manner as in bare PC applications [16]. The intertwining of HTTP and TCP protocols in a Web server is shown in Figure 1. The intertwined HTTP/TCP protocols can also be split after receiving the GET request by partitioning a single server into two servers consisting of a connection server (CS) and a data server (DS) [2]. In this case, the CS establishes the TCP connection and has two-way communication with the client whereas DS communication with the client is only one-way (i.e., it only sends data to the client). The CS also sends an inter-server packet to the DS to communicate the client's request and its state. The CS is connected to the client throughout processing of a given request. In this architecture, one CS interfaces with one or more DSs to provide client services and thus becomes a bottleneck in a mini-cluster configuration [3].

To address such a bottleneck, we split the protocol at an architectural level. This results in a modified client/server architecture, where establishing connections, and data transfers and connection closing are separated entirely. Initially, clients communicate only with the CS and are unaware of the location of the DSs. Then, a DS can completely take over the request from the CS and inform the client. This separation of connection establishment and subsequent data transfer/closing provides increased security at a server level. In this architecture, when a connection is established between a client and a CS, the CS will terminate its connection processing by sending an inter-server packet to a DS and the DS will finish the rest of the session by sending the data and closing the connection. The resulting client/server interactions are shown in Figure 2, where the client sends SYN, ACK-SYN-ACK and GET messages to the CS, and the CS only sends the SYN-ACK and GET-ACK messages to the client. After the CS establishes the connection and sends the GET-ACK, it sends an inter-server packet to a DS and removes itself from further processing of this request. The sending of DATA, ACKs and closing FIN-ACKs will be done by the DS. This enables the CS to be freed completely from the HTTP request and the TCP connection after the GET is processed.

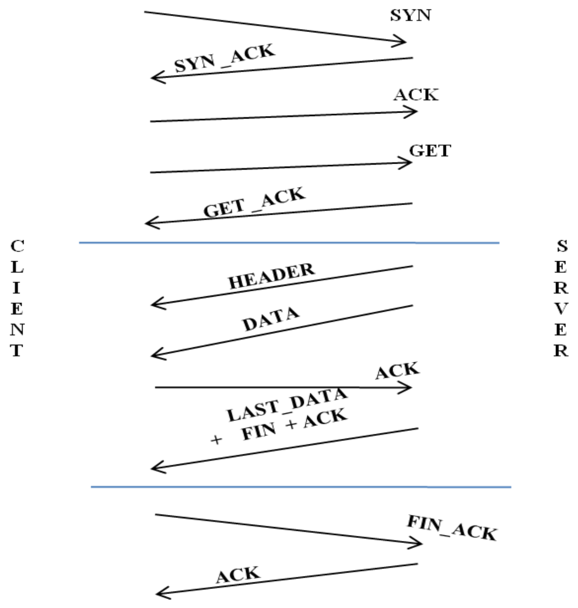

Figure 1. Intertwined HTTP/TCP protocols

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. In protocol splitting, clients can be located anywhere on the Internet. However, there are security and firewall issues that arise when deploying split servers in an Internet if a CS and a DS are on different networks [2]. We have therefore tested the splitting concept with the servers located in a LAN that consists of multiple subnets connected by routers. Splitting and this modified client/server architecture can also be implemented in principle on an OS-based system. However, it is more convenient and much simpler to implement the architecture on a bare PC with no kernel or OS running on the machine.

The rest of the paper is organized as follows. Section II presents related work; Section III describes the split design and implementation; Section IV presents experimental results; Section V discusses the impact of this split architecture; and Section VI contains the conclusion**.**

II. RELATED WORK

Bare PC applications use the Bare Machine Computing (BMC) paradigm, which was previously called dispersed OS computing [19]. That is, there is no OS or centralized kernel running in the machine. Instead, the application is written in C++ and runs as an application object (AO) [20] by using its own device drivers and interfaces to the hardware [18]. The BMC paradigm resembles approaches that reduce OS overhead and/or use lean kernels such as Exokernel [5,11], IO-Lite [22], Palacio [14], Libra [9], factored OS [7], bare-metal Linux [21], and TinyOS [13]. However, there are significant differences such as the lack of centralized code that manages system resources on behalf of applications and the absence of a standard kernel-based TCP/IP protocol stack and socket interface. In essence, the AO itself manages the CPU and memory, and contains lean versions of the necessary protocols. The BMC paradigm also facilitates protocol intertwining, which is a form of cross-layer design. Further details on bare PC applications and bare machine computing (BMC) can be found in [16, 17].

Splitting protocol at a client/server architectural level is different from previous approaches based on migrating TCP connections (or processes or Web sessions), splicing TCP connections or masking failures in TCP-based servers. For example, in migratory TCP (M-TCP) [15], a TCP connection is migrated between servers with client involvement and a new TCP connection. Similarly, in process migration [6], an executing process is transferred between machines, and in proxy-based session handoff [10], a proxy is used to migrate Web sessions in a mobile environment. Protocol splitting is also different from TCP splicing [1] in which two separate TCP connections are established for each request, and from fault-tolerant TCP (FT-TCP) [8] in which a TCP connection continues after a failure enabling a replicated service to survive.

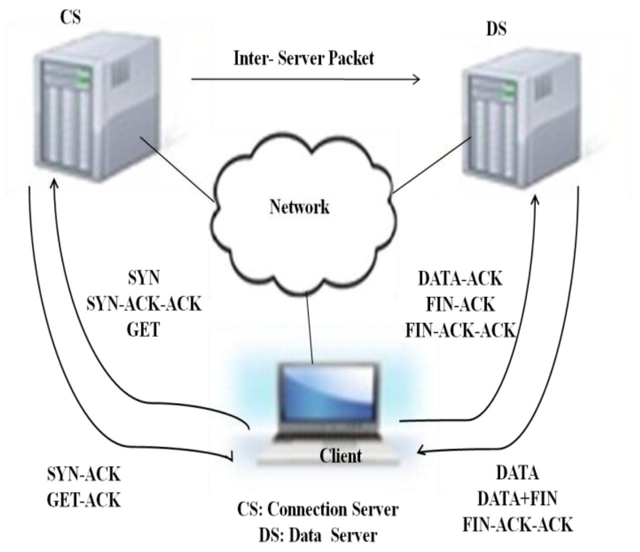

Figure 2. Modified split protocol architecture

III. DESIGN AND IMPLEMENTATION

The modified split protocol client/server architecture differs considerably from that of a conventional client/server system. To demonstrate its feasibility and study performance, we have designed and implemented a client and servers on bare PC systems. The absence of an OS or kernel running in the machine makes it simpler and easier to make the necessary modifications compared to OS-based protocol implementations. Figure 3 shows the high-level design of a client and server in a bare PC. Each client and server updates a TCP state table (TCB) that contains the state of each request. Each TCB entry is made unique by using a hash table with key values determined by an IP address and port number. The CS and DS TCB table entries are referenced by IP3 and Port# (the client's IP address and port number respectively). Similarly, the TCB entry in the client is referenced by IP1 (server IP) and Port#.

The TCB table is the key system component in the client and server design. An entry in this table maintains complete state and data information for a given request. A typical entry consists of about 160 bytes of connection information and another 160 bytes that provide trace, error, logs, and miscellaneous control information. A TCB entry is independent of the machine and can be easily migrated to another PC that runs at a different location. This is not same as process migration [6] as there is no process information contained in the entry. The information in the inter-server packet is derived from the TCB entry and sent to a DS when a GET message arrives from the client.

The client uses IP1 and Port# to address the TCB entry. This requires that the DS use IP1 as its source address when sending data and other packets to the client. However, a client must be aware of both server addresses IP1 and IP2 since they are needed for different purposes. The client knows the CS address IP1 through its original request and by resolving the server's domain name. To inform the client of the DS address IP2 address that it should use on outgoing packets during data transmission, it is included in the HTTP header using a special field. In this architecture, a client could get data from any unknown DS and learn its IP address from the initial data it receives (i.e. the HTTP header). This mechanism simplifies the design and implementation of the split protocol client/server architecture. It also allows the CS to distribute its load among DSs based on their CPU utilization without resorting to complex load balancing techniques [4].

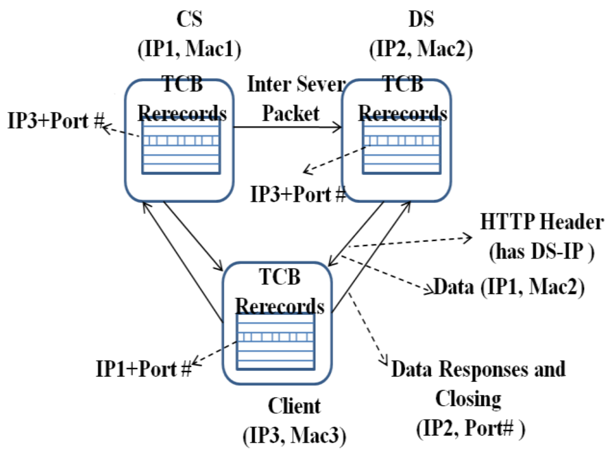

Figure 3. Design structure

It is assumed here that the DS's network allows it to send packets using the address of CS as source. This may be acceptable if the CS and DS are in the same subnet. In general, such IP spoofing could be avoided if the CS informs the client to switch to the IP address of the DS (using the GET-ACK for example). Then the client would change the IP address used to reference the TCB entry of the server from IP1 to IP2, and the DS would send data using its own IP address IP2 as source. For this to be done correctly the TCP connection to IP1 would be closed and a new TCP connection to IP2 would be established by the client. This would increase the overhead, which is avoided by using this approach. For simplicity, the preceding discussion ignored NAT.

We have modified existing bare PC server designs to create the servers and the client. 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 since the connection logic is removed. To implement the bare PC client, the server is modified by swapping its logic with respect to sending and receiving. It is also necessary to implement the client request generator logic in addition to the server logic.

The bare PC server applications do not use any OS-related libraries or system calls. However, the application itself is developed using a standard MS Windows environment and written in Visual C++ (without any *.h files) and MASM Assembler. Most of the direct hardware interfaces are implemented in C/assembly language using software interrupts. The size of the assembly code is approximately 1,800 lines. These direct hardware interfaces include display, keyboard, timers, task management, NIC, and real/protected mode switching. The Intel 82540EM NIC driver code is approximately 3100 lines of C/C++ and 43 lines of assembly code. Similarly, the USB driver uses approximately 133 lines of assembly code; the rest of the code is written in C. The code implementing the Web server is written in C++ in an object-oriented manner. The size of the source code is approximately 22,452 lines of code (not including comments) and 13,744 executable lines. This yields a single monolithic executable AO consisting of 344 sectors of code size 176,128 bytes, which is placed on the USB. The code implementing the Web client is similar, but about 5% larger. The same USB also contains boot code and other user interfaces to load and run the program on a bare PC.

IV. Experimental Results

A. Experimental Setup

The experiments were conducted using 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 Web clients capable of generating 5700 requests/sec were used to create the server workload.

B. Configurations

Figure 4 shows a general configuration for connecting one CS and one or more DSs and clients. The requested file size was 64K. The CS delegates all its requests to one or more DSs for data processing.

C. Measurements: 1-4 DSs, Performance

Figure 5 shows measurements conducted using a single CS and respectively 1-4 DSs. A minimum configuration of a split protocol client server system consists of a single CS and DS. This system can process 900 requests/sec for 64K files. When the number of DSs is increased, performance improves linearly. Figure 6 shows CPU utilization for the CS and DS. CS utilization also increases linearly to 20% for 4 DSs. The DS utilization is maximized as we stressed the server to conduct this experiment.

We expect CS performance to increase linearly until it gets saturated. This is to be expected since 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 can be estimated to be 15 by extrapolation. In an earlier study for splitting based on the usual client/server architecture, it is shown that one CS can support up to 4 DSs before it gets saturated [3]. This implies that the modified client/server architecture can scale up to 15 DSs. Such clusters can potentially be used to build large server clusters as in [12].

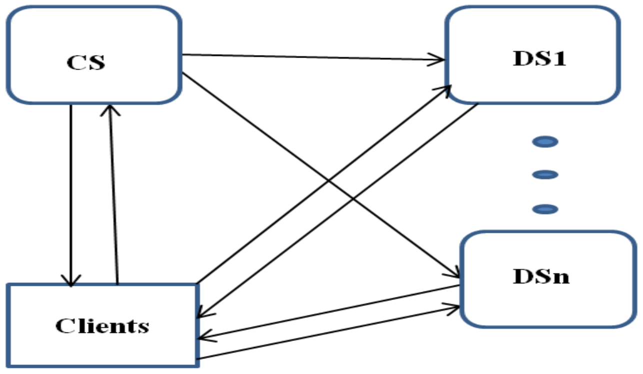

Figure 4. Split protocol configuration

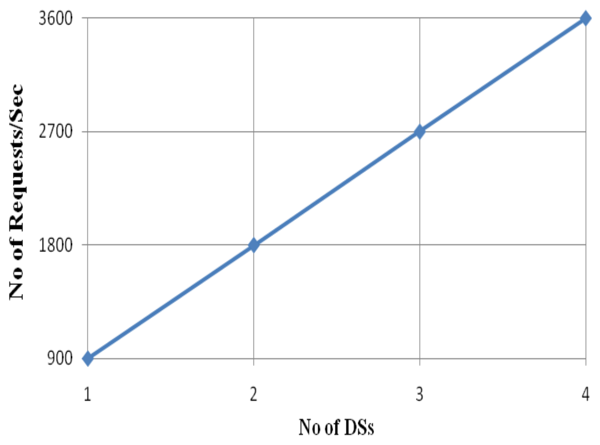

Figure 5.Throughput

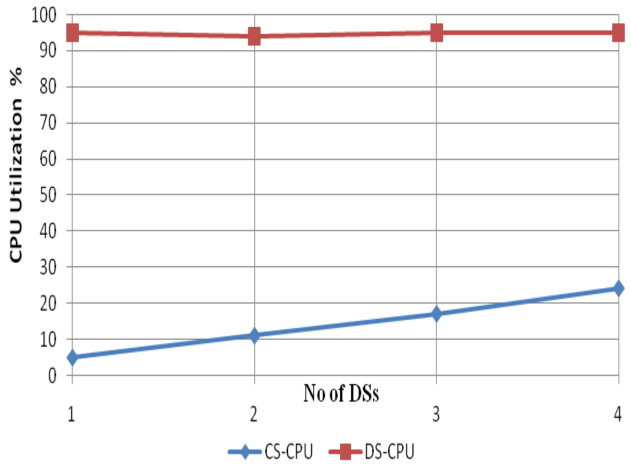

Figure 6. CS and DS CPU Utilization

D. Measurements: 1-4 DSs, Actual Times

Figure 7 shows the actual time taken by a client/server configuration for processing 1.62 million requests. As expected, these results indicate that the amount of time taken to process all the requests is inversely proportional to the number of DSs added to the system. Similar performance gains will be difficult to achieve using a conventional server cluster since it requires external load balancing techniques that would increase the overhead. The proposed CS/DS cluster manages the load balancing within the CS itself and eliminates this overhead. In addition, partitioning the load for DSs at the CS level is simpler since the latter communicates with all its DSs. In our experiments, a round robin approach was used to delegate requests incurring no penalty in load distribution.

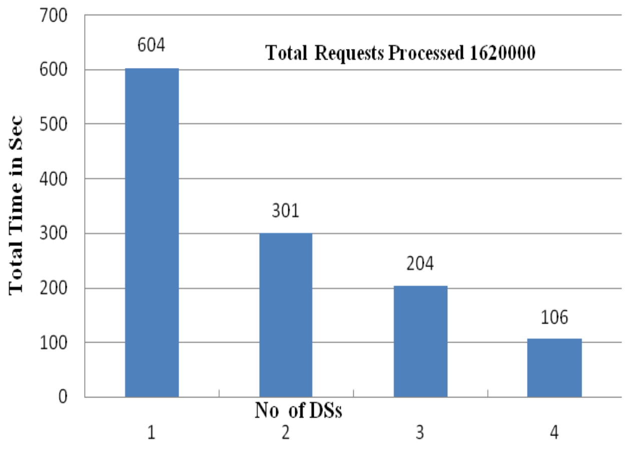

Figure 7. Processing time

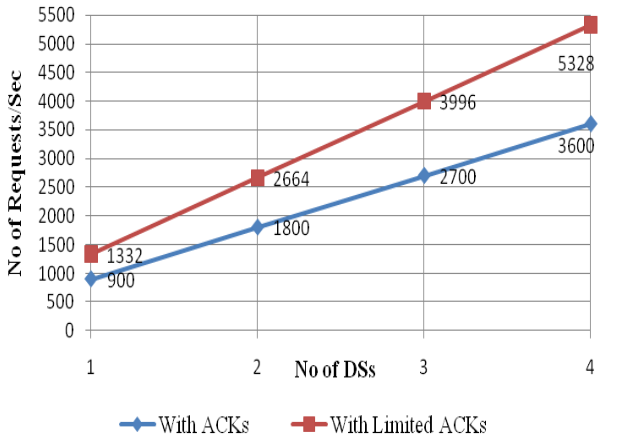

Figure 8. Throughput with varying ACKs

E. Measurements: 1-4 DSs, Varying Acks

In order to improve DS performance further, we reduced the number of ACKs for data by modifying the client code. Instead of sending ACKs for each data packet, we sent a single ACK for all the data and another for closing the connection (FIN-ACK-ACK). This is similar to sending a negative ACK (NAK) only when data is not received. By reducing data ACKs to a minimum, we measured the performance for one to four DSs as shown in Figure 8. The figure also shows the throughput for normal ACKs. It can be seen that limiting the number of ACKs for data improves performance by 48% at the DS level.

Figure 9 shows the CPU utilization for the CS and DSs with varying ACKS. The DS utilization has peaked due to the maximum load requested by the clients. The CS utilization also increases with limited ACKs because the CS is now 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 extrapolation.

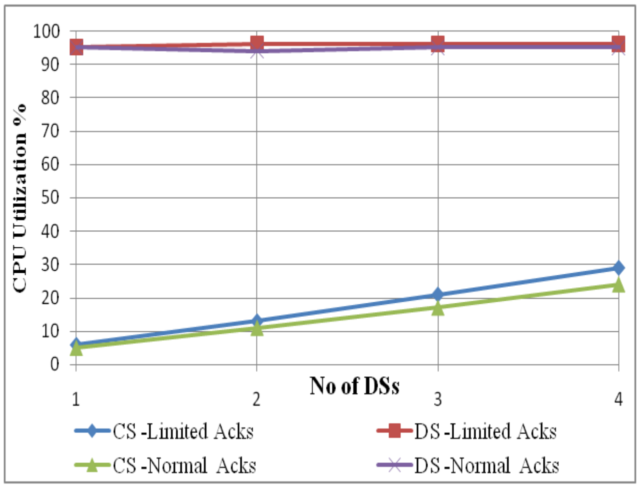

Figure 9. Utilization with varying ACKs

Thus, a typical CS-DS cluster may contain up to 13 DSs with limited ACKs and 15 DSs with normal ACKs. Figure 9 shows that the CPU utilization increased from 24% to 29%, for 4 DSs due to handling additional load. Thus, the 48% improvement shown for 4 DSs in Figure 8 may not continue when there are a large number of DSs in the system.

F. Lack of comparison with OS based systems

We were unable to compare performance of the modified client/server split protocol architecture for bare PC servers and clients with that for OS-based systems, as it is very difficult to implement split protocols on them. Most OS-based systems do not provide easy access to split the protocol. In contrast, it is easy to access and modify the protocols in bare PC applications.

V. IMPACTS OF SPLITTING

Splitting enables the functionality of a protocol to be split across machines or processors. In particular, splitting the TCP protocol requires modification to the usual client/server architecture. Some of the impacts of splitting are as follows:

- Split protocol configurations based on connections and data can be used for constructing large server clusters (4-15 DSs)
- Scalable performance can be achieved by adding DSs to the cluster without incurring load balancing overhead
- A uniform reduction in processing time can be achieved by adding additional DSs as they work independently and concurrently
- Complex load balancing techniques and dispatchers are not needed
- Connections and data transfers can be completely separated (this may provide additional security due to data server isolation)
- Connection and data servers can be located in different places; for example, data servers can be located in close proximity to data (this raises some network-related issues as noted earlier)
- Client connections can be easily monitored without interrupting client data communication
- Server designs can be simplified; the CS design in particular is simpler and easier to manage
- This approach can also be used for database servers and file servers.

VI. CONCLUSION

We studied a modified client/server computing system with protocol splitting across connection and data servers. This approach differs from conventional client/server design and implementation in that the data servers alone handle communication after a connection is initially set up by the connection server. The server architecture is scalable and allows large server clusters to be built. We discussed design and implementation details using clients and servers running on bare PC systems. The experimental results showed performance improvements of up to 48% by increasing the number of DSs and limiting the ACKs needed for data transmission.

We also discussed the impacts of splitting. When evaluating the tradeoff of splitting versus non-splitting, it is necessary to consider the overhead and cost of load balancers and dispatchers, which will result in lower throughput and increased response times. This study used only bare PC systems; we were unable to provide a comparison with conventional systems due to the difficulty of implementing protocol splitting on OS-based clients and servers. It would be useful to investigate the benefits of splitting using OS-based Web servers and applications. More studies are also needed to evaluate the security benefits of split server clusters, and their scalability and performance using a variety of workloads.

ACKNOWLEDGEMENT

We sincerely thank NSF and in particular the late Dr. Frank Anger, who initially supported this work by funding through SGER grant CCR-0120155. Without his encouragement, bare machine computing concept and explorations could not have been possible.

REFERENCES

[1] 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.

[2] 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.

[3] B. Rawal, R. Karne, and A. L. Wijesinha. "Mini Web Server Clusters for HTTP Request Splitting," 13th International Confrence on High performance Computing and Comunication, HPCC-2011, Banff, Canada, Sept 2-4,2011.

[4] Ciardo, G., A. Riska and E. Smirni. EquiLoad: A Load Balancing Policy for Clustered Web Servers". *Performance Evaluation*, 46(2-3):101-124, 2001.

[5] D. R. Engler and M.F. Kaashoek, "Exterminate all operating system abstractions," Fifth Workshop on Hot Topics in operating Systems.

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

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

[8] 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, May 2009, Volume 27, Issue 2, Article 4.

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

[10] 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.

[11] 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)**,** February, 2002, vol. 20, Issue 1, pp. 49 – 83.

[12] L. A.Barreso, J. Dean, and U. Holzle, Web Search for a Planet: The Google Cluster Architecture, IEEE Micro, March 2003.

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

[14] J. Lange, K. Pedretti, T. Hudson, P. Dinda, Z. Cui, L. Xia, P. Bridges,S. Jaconette, M. Levenhagen, R. Brightwell, and P. Widener. Palacios and Kitten: High performance operating systems for scalable virtualized and native supercomputing. Technical report, EECS Northwestern University, July 2009.

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

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

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

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

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

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

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

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