TOWSON UNIVERSITY
OFFICE OF GRADUATE STUDIES

INTEGRATION OF A STORAGE SYSTEM FOR BARE MACHINE COMPUTING

by

Hamdan Ziyad Alabsi

A Dissertation

Presented to the faculty of

Towson University

in partial fulfillment

of the requirements for the degree

Doctor of Science

Department of Computer & Information Sciences

Towson University

Towson, Maryland 21252

May, 2018

© 2018 by Hamdan Alabsi

All Rights Reserved

TOWSON UNIVERSITY
OFFICE OF GRADUATE STUDIES

DISSERTATION APPROVAL PAGE

This is to certify that the dissertation prepared by Hamdan Alabsi, entitled "INTEGRATION OF A STORAGE SYSTEM FOR BARE MACHINE COMPUTING," has been approved by the dissertation committee as satisfactorily completing the dissertation requirements for the degree Doctor of Science in Information Technology.

Ramesh K. Karne — 5-4-18

Chair, Dissertation Committee, Dr. Ramesh K. Karne — Date

5/4/18

Committee member, Dr. Alexander L. Wijesinha — Date

5/14/2018

Committee member, Dr. Chao Lu — Date

Subrata Acharya — 5/4/2018

Committee member, Dr. Subrata Acharya — Date

Janet V DeLany — 5-10-18

Dean of Graduate Studies — 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 Karne (chair), Dr. Alexander Wijesinha, Dr. Chao Lu and Dr. Subrata Acharya for supporting this research. I am especially thankful to Dr. Karne and Dr. Wijesinha for all the long hours in the lab especially on many weekends, and for their support and valuable advice.

I offer my heartfelt thanks to my beloved parents Ziyad and Mansurah ,my brothers, Emad, Feras, and my sisters, Maram, Marwah and Mawdah , whose words have encouraged me to complete my doctoral study.

To my colleagues in Towson University ;Majed Aljazaeri, Mohammed Alyami, Faris Almansour , Rasha Almajed and William Thompson I say thank you for all the help and intimacy we shared during the years of doctoral research. I also would like to thank Dr. Sidd Kaza, Chair of the Department of Computer and Information Sciences, and Dr. Janet V. Delany Dean of Graduate Studies at Towson University, for facilitating this work. I am also grateful to the late Frank Anger (National Science Foundation) for his support of the Application Oriented Object Architecture (AOA), which evolved into Bare Machine Computing research and consequently made this dissertation possible.

Abstract

INTEGRATION OF A STORAGE SYSTEM FOR BARE MACHINE COMPUTING

Hamdan Alabsi

This research investigates and develops strategies for integrating a storage system with bare machine computing (BMC) applications, which run without the support of any operating system (OS) or kernel. Any storage system requires reliability, availability, survivability and high performance. We first explored reliability and performance of storage data using a redundant array of independent disk (RAID) technique and applied this to BMC file systems. The RAID design and implementation was done using 2, 4, and 8 detachable mass storage devices (USB flash drives). We resolved many design issues that arose when integrating a file system with multiple flash drives, and conducted experiments on a variety of storage data split configurations. We then integrated the file system and RAID application with a bare PC Web server that allows users to access storage online.

We also integrated the SQLite database with the bare PC file system and the bare PC Web server. We showed that the BMC architecture allows us to integrate other components. We used the integrated database system to demonstrate two popular applications that run on a bare PC. The database provides services for clients via a Web interface and associated queries. It is further used to provide an email service for a select group of clients, where the database serves as storage media for messages. As the email service is limited to a small group of users, it is a closed and secure communication system.

In this research, we integrated many components including the Web server, database and email applications, file system, RAID, and SQLite enabling them to run as a single monolithic executable on a bare machine. The integrated BMC system has inherent security and performance benefits due to not running an OS or kernel. Our work provides a foundation to build future BMC systems that integrate additional components with bare PC applications.

Table of Contents

List of Tables ... viii

List of Figures ... ix

1 INTRODUCTION ... 12

2 RELATED WORK ... 15

- 2.1 Bare Machine Computing Background ... 15
- 2.2 BMC Applications ... 17
- 2.3 Other Related Work ... 18

3 INTEGRATION OF RAID IN A BARE PC FILE SYSTEM ... 19

- 3.1 System Architecture ... 19
- 3.2 Design and Implementation ... 22
- 3.3 Performance ... 26
  - 3.3.1 File Size ... 26
  - 3.3.2 Block Size ... 29
  - 3.3.3 Number of USBs ... 30
- 3.4 Summary ... 31

4 INTEGRATION OF SQLITE IN A BMC WEB SERVER ... 32

- 4.1 System Overview ... 33
- 4.2 System Integration ... 35
- 4.3 Design Insights ... 39
- 4.4 Implementation ... 44
- 4.5 Functional Operation ... 45
  - 4.5.1 Server Operation ... 45
  - 4.5.2 Database Application ... 48
  - 4.5.3 Database Mail Application ... 52

5 SIGNIFICANT CONTRIBUTIONS ... 56

6 CONCLUSION ... 57

APPENDIX ... 58

A. Compilation Environment ... 58

B. Operating Environment ... 58

C. Directory Structure ... 74

D. USB Map ... 76

E. Memory Map ... 77

F. PHP Files ... 80

7 References ... 90

Curriculum Vitae ... 96

List of tables

Table 1. Comparison of Conventional and BMC Computing 16

Table 2. Number of Blocks for RAID0, RAID1, RAID5 Using A 64 MB File Size .. 27

Table 3. PHP Files and their Parsed Attributes.. 42

LIST OF FIGURES

Figure 1. Bare RAID Architecture 21

Figure 2. A 4 MB File with 1024 Sectors/block 22

Figure 3. File Control Structure 23

Figure 4. RAID Control Structure 24

Figure 5. Memory Map 24

Figure 6.Write Times for File Sizes between 4 to 64 MB. 27

Figure 7. Read Times for File Sizes between 4 to 64 MB. 28

Figure 8. Flush Times for File Sizes between 4 to 64 MB 29

Figure 9. RAID5 Write and Read Times for 128 to 1024 Sectors/Block 30

Figure 10. RAID0 Write and Read Times Using Multiple USBs (64 MB File, 1024 Sectors/Block) 31

Figure 11. System Architecture 34

Figure 12. Task Interface 35

Figure 13. Integration of SQLite System 37

Figure 14a. Network Flow Control 40

Figure 15. PHP File 43

Figure 16. Server SQLite Interface 46

Figure 17. A Wireshark Trace for GET and The Client Page 47

Figure 18. A Wireshark Trace for Read and The Read Mail Page 48

Figure 19. Database Access Page 49

Figure 20. Login Page 49

Figure 21. Menu Screen 50

Figure 22. Query Page 51

Figure 23. Query Results Page 51

Figure 24. Send Mail Page 53

Figure 25. Receive Mail Page 54

Figure 26. Receive Mail Results Page 54

Figure 27. Delete Page 55

Figure 28. Process Request Part 1 59

Figure 29. Process Request Part 2 60

Figure 30. Process Request Part 3 61

Figure 31. Process Request Part 4 62

Figure 32. Process Request Part 5 63

Figure 33. Process Request Part 6 64

Figure 34. Shell.c Part 1 66

Figure 35. Shell.c Part 2 67

Figure 36. Shell.c Part 3 68

Figure 37. Shell.c Part 4 69

Figure 38. Shell.c Part 5 70

Figure 39. Shell.c Part6 71

Figure 40. Shell.c Part7 72

Figure 41. Shell.c Part8 73

Figure 42. Directory 75

Figure 43. USB Map 76

Figure 44. Memory Map Part 1 78

Figure 45. Memory Map Part 2 79

Figure 46. b-login.php Part1 81

Figure 47. Inbox.php Part 1 83

Figure 48. Inbox.php Part 2 84

Figure 49. Sendmail.php Part1 85

Figure 50. Sendmail.php Part 2 86

Figure 51. b_compose.php Part 1 87

Figure 52. b_compose.php Part 2 88

Figure 53. b_compose.php Part 3 89

1 INTRODUCTION

The operating system (OS) is a fundamental software element that is needed to run most computer applications. An OS manages and coordinates computer hardware and software resources and provide services to the users. However, an OS creates complexity, introduces security vulnerabilities and degrades performance due to its many functions and features. OS based systems are inherently layered and open systems. As new versions of OSs emerge, existing applications may need to be ported to the new platforms resulting in obsolescence and waste. An alternate approach to developing computer applications is to use the bare machine computing (BMC) [1] paradigm that reduces obsolescence and waste. In the BMC paradigm applications run without the support of any operating system (OS) or centralized kernel. The BMC approach optimizes and improves the system with respect to performance and security. It also eliminates system complexity that is due to the OS environment.

The first part of this dissertation explores the reliability and performance of file systems. Most file systems and redundant array of independent disks (RAID) architectures depend on an underlying OS platform or kernel. We design and implement a novel RAID file system for USBs that runs on a bare PC without using any operating system or kernel. We present preliminary performance data that measures the time for file operations such as read and write for various settings of configuration parameters such as file size, sectors per block and number of USBs. In addition to providing protection against OS-related security vulnerabilities, bare RAID systems have the advantages of privacy and portability.

The second part of this dissertation investigates the integration of a file system, Web server and SQLite [2]. Existing SQLite is a server-less database engine, in which read and

write access is done directly using database files on disk. There is no intermediary server process [3]. Previously, an amalgamated SQLite was transformed to run on a bare PC as shown in [4] [5]. This transformation process required the removal of conventional OS system calls and their replacement with direct hardware interfaces designed for bare machine computing (BMC) [1]applications. The standard system calls for memory were replaced with bare PC memory allocation interfaces. As SQLite provides a virtual file interface (VFS), it is replaced with a bare PC file system interface (using FAT32) [6] [7]. The transformation and integration of SQLite into bare PC applications was done without understanding and changing the existing code in SQLite [4] [5]. Our work enables SQLite to be integrated with a bare PC Web server and file system in a client/server environment.

The SQLite amalgamated package is written in C and bare PC applications are written in C++. Thus, there were special interfaces needed to communicate between C and C++ programs. Integration of standalone SQLite into a client/server environment also poses many challenges as described in this dissertation. SQLite running as a client/server system can be used by Web clients for database creation and access. The database will be stored on a USB flash drive and treated like any other database. The USB flash drive can be secured assuming that physical security is controlled by a given user. The SQLite database management system runs on a bare PC with a bare PC file system. In addition, SQLite is also used as a database mail (DB mail) system for a select group of users. Although it is Web-based, DB mail does not interoperate with other conventional mail systems. It provides closed communication between selected users to provide a highly secure system. As the system is closed to other users, many existing email problems such as spam and security vulnerabilities are avoided.

The system is especially suited for secure communication among users in the defense sector, government agencies and private enterprises. The rest of the dissertation provides details of RAID and SQLite integration in a BMC system.

2 RELATED WORK

In this section, we present an overview of the BMC paradigm and existing bare PC applications. We also discuss other approaches that attempt to minimize OS overhead.

2.1 Bare Machine Computing Background

BMC is a programming paradigm based on bare machines. In the BMC paradigm, applications run without the support of any operating system (OS) or centralized kernel i.e., no intermediary software is loaded on the bare machine prior to running applications. The applications, which are called bare machine applications or simply BMC applications, do not use any persistent storage or a hard disk, and instead are stored on detachable mass storage such as a USB flash drive. A BMC program consists of a single application or a small set of applications (application suite) that runs as a single executable within one address space. BMC applications have direct access to the necessary hardware resources. They are self-contained, self-managed and self-controlled entities that boot, load and run without using any other software components or external software. BMC applications have inherent security due to their design. There are no OS-related vulnerabilities, and each application only contains the necessary (minimal) functionality [8]. There is no privileged mode in a BMC system since applications only run in user mode. Also, application code is statically compiled-there is no means to dynamically alter BMC program flow during execution.

In many ways, the BMC paradigm differs from conventional computing. There is no centralized kernel or OS running during the execution of BMC applications. Also, a bare machine in the BMC paradigm does not have any ownership or store valuable resources; and it can be used to run general purpose computing applications. Such characteristics are

not found in conventional computing systems including embedded systems and system on a chip (SOC). Table 1 compares conventional computing and bare machine computing [1].

OS-Based System	Bare Machine System
OS, embedded OS, or Kernel	No OS, No embedding, No Kernel
Centralized Control	Application Control
Open Systems	Closed Systems
DLLs, Open Ports	No DLLs, No Open Parts
Mass Storage on-board	No mass Storage on-board, External Storage Only
Multiple Modules, Software, Other vendor entities	Single application suite or a module
Dynamic Binding	Static Binding
Vulnerable to OS and other commercial software	No OS or Commercial Software vulnerabilities
General purpose, Multi-user access	Application centric and Single user controlled (or Group controlled)
Application depend upon other entities during execution	Self-controlled, Self-executed, Self-managed application
Many resources to be exploited by intruders In the box	No valuable resources In the bare box
Virus, worm, etc. can get in	None can get in
Leaves execution trace or valuable resources	No valuable resources after execution
Open communication to external users	Closed communication to un-intended users
Box has to be secured	Box can be used by any one
Privilege and user modes	User mode only
Other users can access the system while running	Only intended users can access while running through network interfaces
Intruders can damage resources	Intruder may get the system go down but cannot damage any resources
Needs porting to new environments	No porting, runs on any x86(or target) architecture
Frequent updates, dumping and wastage	No frequent updates, no dumping and no wastage
Applications & System programs	Only application programs

Table 1. Comparison of Conventional and BMC Computing

2.2 BMC Applications

Several complex bare applications have been developed in the bare machine computing laboratory at Towson University, and most were the most outcome of doctoral research work. Long He [9] 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. Khaksari [10] developed the first VoIP soft-phone that runs on a bare PC and provides secure communication on an end-to-end basis. Alexander [11] built a SIP server and a bare SIP user agent to demonstrate the feasibility of running high performance SIP servers with secure communication using the SRTP protocol. Ford built the first Email server that runs on a bare PC and provides compatible performance to related commercial email servers [12] [13]. Emdadi [14] implemented the complex TLS protocol for a bare Web server. Yasinovskyy [15] implemented the IPv6 protocol for a bare PC VoIP softphone client. Tsetse implemented the bare PC NAT Box and 6to4 Gateway [16] [17] [18]. IPSec on a bare PC was implemented by Kazemi [19]. Rawal [20] [21] developed a unique split protocol concept and applied it to Web servers that run on a bare PC. He also developed mini-cluster configurations for high-performance bare PC Web servers based on the split protocol concept [22]. Appiah-Kubi developed a secure Webmail server using TLS [23]. Karne [24] developed a USB mass storage device driver for BMC applications. Loukili [25] [26] studied TCP performance in BMC, and Augusto-Padilla [27] studied the possibility of transforming a Linux wireless driver to run on a Bare PC. Okafor [4] [5] demonstrated a feasibility of transforming SQLite database to run on a bare PC. Alexander [28] [11] showed the applicability of the BMC paradigm to handheld devices. Liang [29] and

Thompson developed FAT32 file systems [7] [6] that run on bare PCs. Chang developed a multicore architecture for running a bare PC applications on multiple cores [30]. These previous doctoral research projects made possible the discovery of many novel system characteristics that are unique to BMC.

2.3 Other Related Work

Reducing OS complexity and moving code to user space has been a research interest for many over a long period of time. There are many other approaches such as Exokernel [31], OS-Kit [32], IO_Lite [33], Palacio and Kitten [34] that move parts of the kernel to the application domain does not eliminate a centralized middleware or control in the system. The BMC paradigm is not the same as the above approaches as it is at the other end of the spectrum. Since there is no centralized OS or Kernel running in a BMC system, the applications have the total control of the hardware.

3 INTEGRATION OF RAID IN A BARE PC FILE SYSTEM

In this section, we describe a USB-based bare PC file system that integrates RAID. We give details of the novel bare PC RAID design and implementation and also conduct experiments to measure system performance.

3.1 System Architecture

There are many RAID configurations and commercial RAID systems such as [35] [36] [37]. They typically require a hard disk and an OS. This thesis implemented RAID0, RAID1, and RAID5 configurations for a bare PC system by writing a bare RAID application that extends the bare file and mass storage system for USBs [6].

Figure 1 shows the bare RAID system architecture. The FAT32 file system, USB 2.0 device driver [24] and RAID object (RAIDObj) are used to implement the bare PC application. The FAT32 specification is used for file implementation as it is simple and easy to structure for BMC applications. Direct bare PC application interfaces are designed to interface with RAIDObj, which can perform operations such as backup, retrieve, check_for_errors, and recreate_failed_data.

Other direct hardware interfaces provide direct communication to hardware for an application programmer. For example, an application program and the programmer controls required memory, display layout, and interfaces such as keyboard, file and network interfaces. The USB driver interfaces are also controlled by the application. There is no privileged or system mode in this system (all bare applications run as user applications since there is no OS). The RAIDObj provides the logic for implementing the bare RAID system. It also interfaces with the existing file system that was designed for the bare PC application. The file system in turn interfaces with the bare PC USB device driver, which

communicates with the hub controller in the PC. In this implementation, the PC has two controllers which provide a total of 10 USB ports. Only 4 ports from each USB controller are used for the design of the bare RAID system.

All bare PC systems have a common task structure that enables execution of a bare application or application suite. A circular list stores tasks using a FIFO (first-in-first-out) policy for processing. A task is removed and inserted into the circular list when it starts, and re-turned to the stack when it ends. Each task runs with-out interruption and only gets suspended when it has to wait for an event. It is then selectively resumed on event occurrence regardless of its FIFO position. Different applications are handled by generating and storing multiple task pools in their corresponding stacks. Tasks are defined and their behavior is controlled within the program according to the needs of a given application.

The event-based programming model eliminates the need for centralized control, synchronization primitives and concurrency control in a single core machine. In a multicore machine, message passing is used instead of shared memory. Bare application programs directly communicate to the hardware by using the hardware API (HAPI). In addition, a standard bare API is provided for tasking, networking, file system, interrupts, memory and other hardware accessories, thus making the low-level internal details transparent to the programmer. However, these hardware interfaces are not the same as conventional system calls, which require interrupts and the ability to distinguish between user and kernel modes. In bare machine computing, there is a single application mode and only one set of applications (i.e., an application suite) runs at a time.

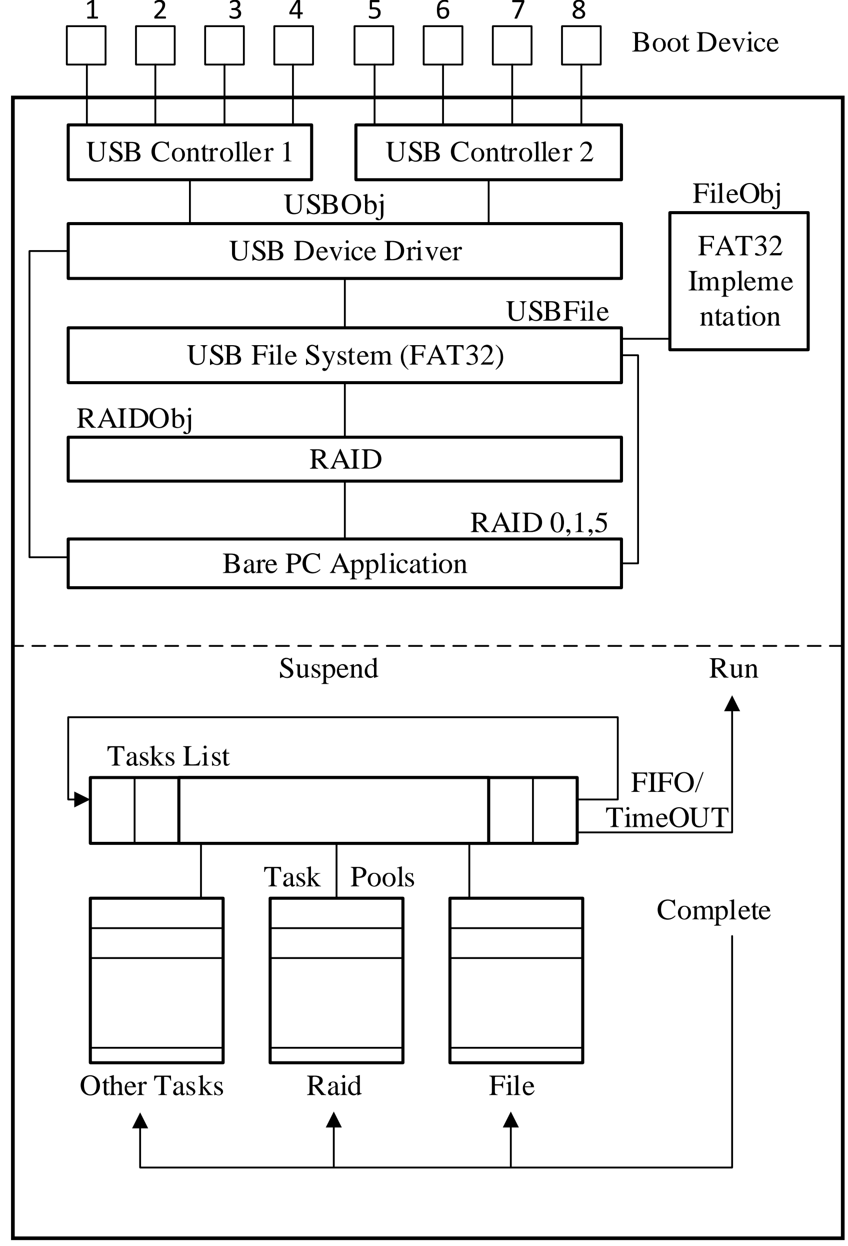


Figure 1. Bare RAID Architecture

3.2 Design and Implementation

RAID0 and RAID1 configurations can use up to 8 USBs, whereas RAID5 always uses 4 USBs. Each RAID configuration splits data differently and RAIDObj determines how the data is organized. Figure 2 illustrates how a 4 MB file splits into 8 blocks (blocks 1– 8) for a 4 USB system with 1024 sectors/block. Each sector is assumed to contain 512 bytes. RAID0 uses block striping, RAID1 uses mirroring and RAID5 uses striping and distributed parity as shown in Figure 2.

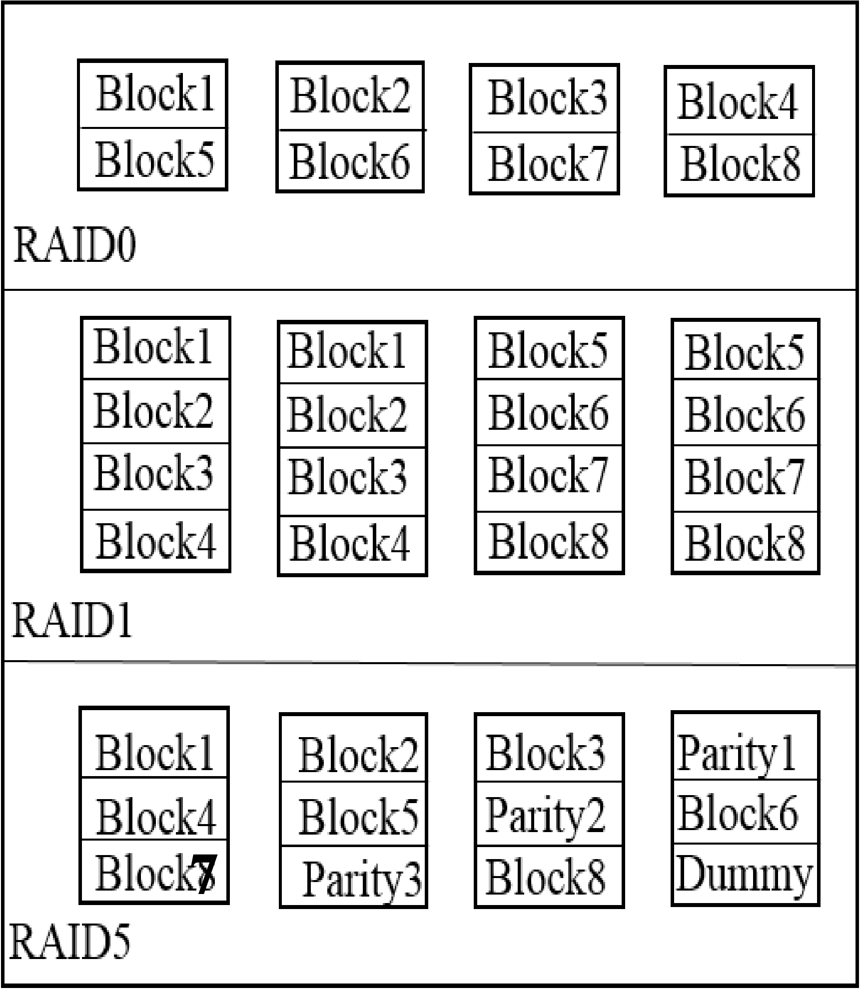


Figure 2. A 4 MB File with 1024 Sectors/block

One USB serves as the bootable device, which contains the BMC RAID application and the file data. The bare application is small and takes a negligible amount of space compared to the drive capacity. There are two primary control structures used in RAID design and implementation. The file control structure (FCS) shown in Figure 3 consists of 96 bytes and contains information needed to manage the file.

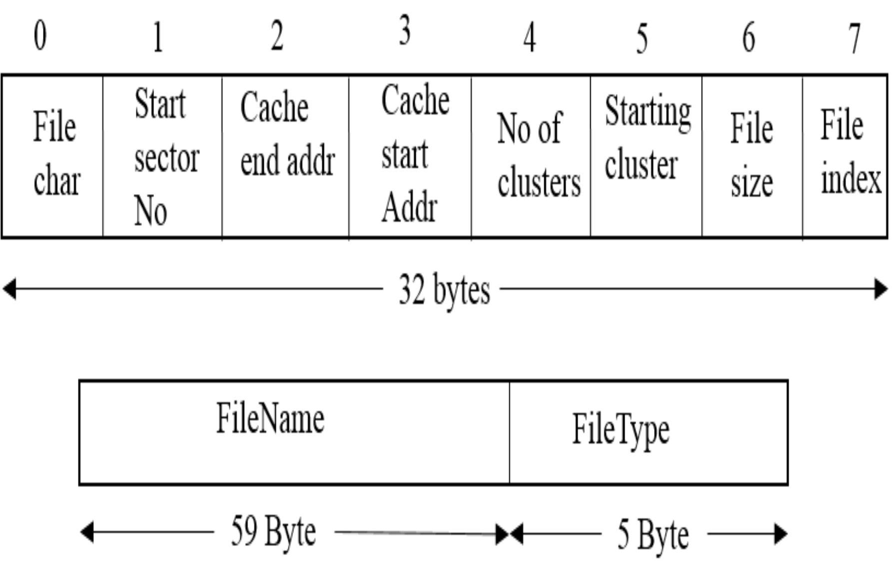


Figure 3. File Control Structure

The FCS structure is not visible to the user. The file is stored on a single flash drive if there is no RAID. If RAID is used, depending upon the RAID type, the file is divided into blocks and stored on multiple USBs. When the file is read, the partitioned blocks are reassembled and the file is constructed. A single block contains multiple sectors. A single block at a time is stored on or retrieved from a given USB. RAID5 uses parity blocks that are placed on different USB drives depending on block location. To simplify RAID5 design, we defined a Raid Control Structure (RCS) consisting of 32 bytes as shown in Figure 4.

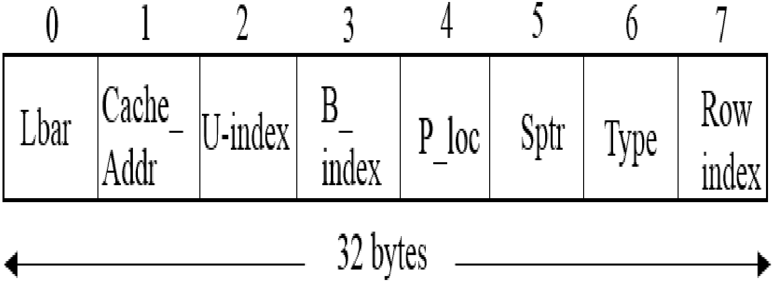


Figure 4. RAID Control Structure

The control parameters needed to manage blocks are stored in the RCS. The temporary memory storage shown in Figure 5 helps to make USB read and write operations efficient. Specifically, it is used to assemble data blocks, generate parity on the blocks and write to USBs. It is also used to check parity and correct blocks as needed before reassembling the file.

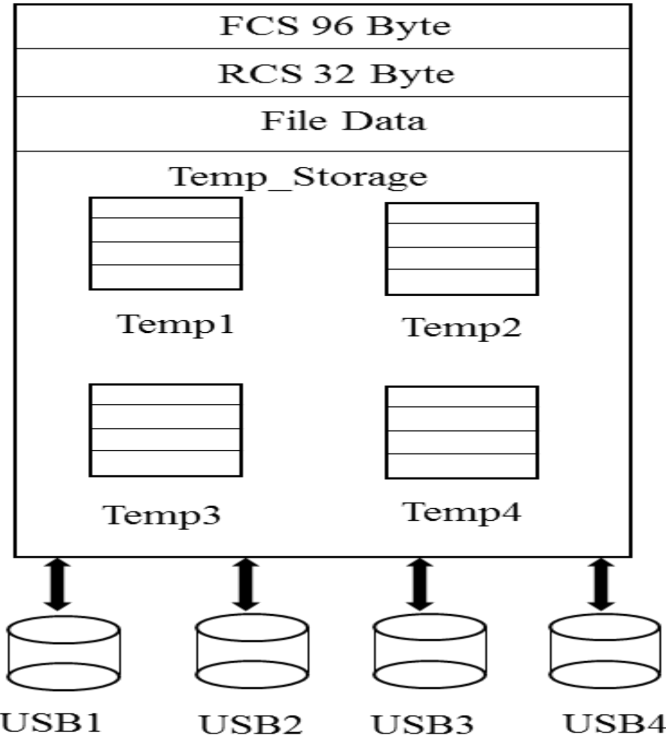


Figure 5. Memory Map

The FCS and RCS structures simplify design and implementation of the RAID system. They may be viewed as schema for the RAID implementation. These structures and the application executable are stored on the USB to make the system independent of any execution environment. That is, a bootable USB can be carried and run on any desktop that has USB ports. At present, the code only runs on any x86 compatible PCs. However, the HAPI concept is not limited to a particular CPU architecture and provides the basis to build bare RAID systems that run on other devices.

The RAID system is written in C/C++. The HAPI is in C++ and its internal implementation uses Microsoft assembly code. The boot and loader code is written in NASM. The USB device driver [24] is primarily written in C except for a small number of lines of assembly code. The bare RAID application can be made available on the Web since it has been integrated with existing code for a bare Web server (as an application suite). The size of the executable containing the USB driver, Web server, file system, RAID, user interfaces and the HAPI is about 785 KB. The bootable USB contains this application suite along with the RAID file. The other data USBs only contain partitioned data necessary for RAID. The data could be encrypted and authenticated using standard algorithms that have been implemented on existing bare systems. For additional security, the bootable and data USBs must be physically secured. We have not used any hardware controls in the USBs for RAID or security.

3.3 Performance

Measurements for a BMC file system and raw read/write measurements for a bare USB driver are given in [6] and [24] respectively. Here, we measure the performance of the bare RAID system with different configuration parameters (file size, sectors/block and number of USBs) by measuring the time for write, read and flush file operations. Table 2 shows the number of blocks stored in each USB for a 64 MB file with 512 sectors/block. RAID1 uses mirroring, so the number of blocks is double the number in RAID0. In RAID5, for each of the 3 data blocks there is one parity block and one or two dummy blocks to make the number of blocks evenly divisible by 4 (86 parity blocks and 2 dummy blocks for a 64 MB file). Compared to RAID0, RAID1 requires twice the number of blocks and RAID5 1.375 times the number of blocks. The results below were obtained using the bare RAID file system using 2GB capacity USBs and a Dell Optiplex 960 desktop with 3 GHz clock speed and 4 GB memory.

3.3.1 *File Size*

Time is measured in milliseconds for write, read and flush operations on each USB when the file size varies from 4 MB to 64 MB. For a 64 MB file using 512 sectors/block for example, the time to write the number of blocks specified is measured as shown in Table 2 to each USB. Figure 6 shows write times for RAID0, RAID1 and RAID5. Write time for RAID1 is 2.17 (3997/1838) times higher than for RAID0, which is slightly higher than the value in Table 2. Similarly, write time for RAID5 is 1.39 (2573/1838) times higher than RAID0, which is also slightly than the value in Table 2 (RAID5 includes times to generate the parity on blocks).

RAID0	**RAID1**	**RAID5**
64	128 (2 times)	88 (1.375 times)

Table 2. Number of Blocks for RAID0, RAID1, RAID5 Using A 64 MB File Size

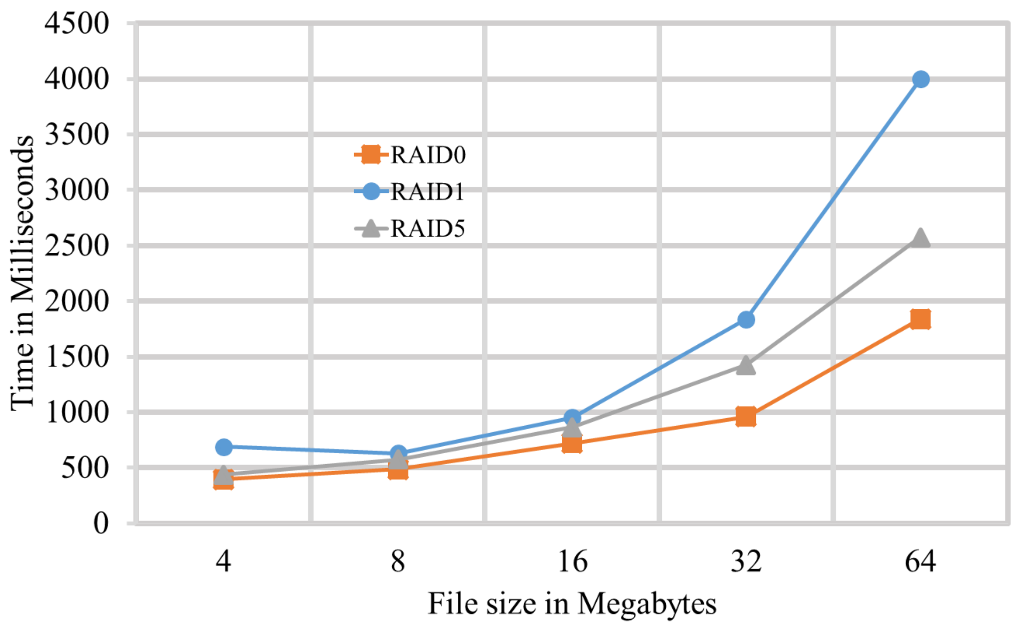


Figure 6.Write Times for File Sizes between 4 to 64 MB.

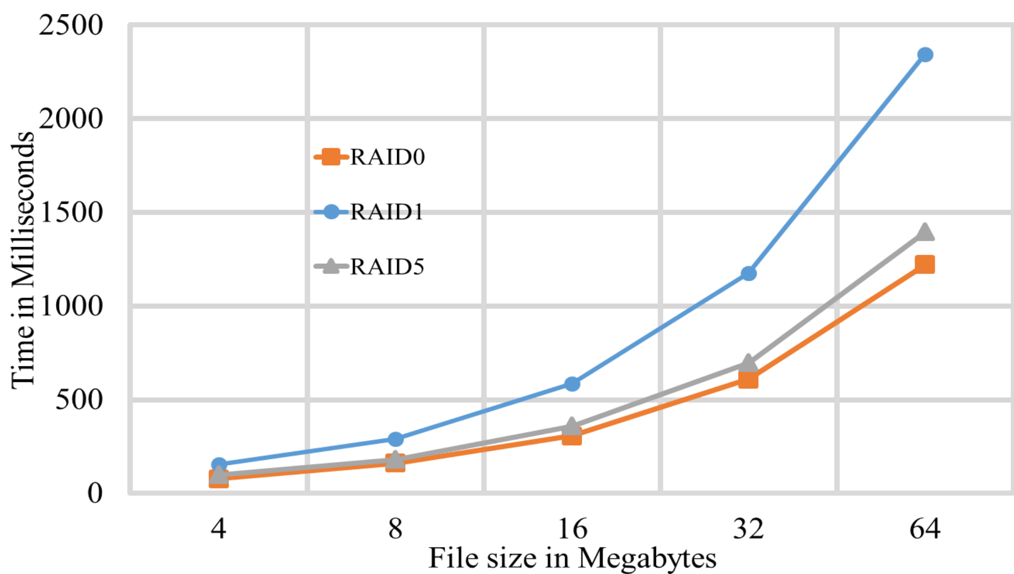


Figure 7. Read Times for File Sizes between 4 to 64 MB.

Figure 7 shows read timings using the same configuration parameters. Read times for RAID1 and RAID5 are respectively 1.92 and 1.15 times more than for RAID0, which is also close to the values in Table 2. Notice that read times are always less than the corresponding write times. The flush operation requires flushing the data from memory to the boot USB (including all data, root directory and FAT tables). Figure 8 shows that the flush times have little variation for each RAID configuration as the file data after reassembly is same as the original data.

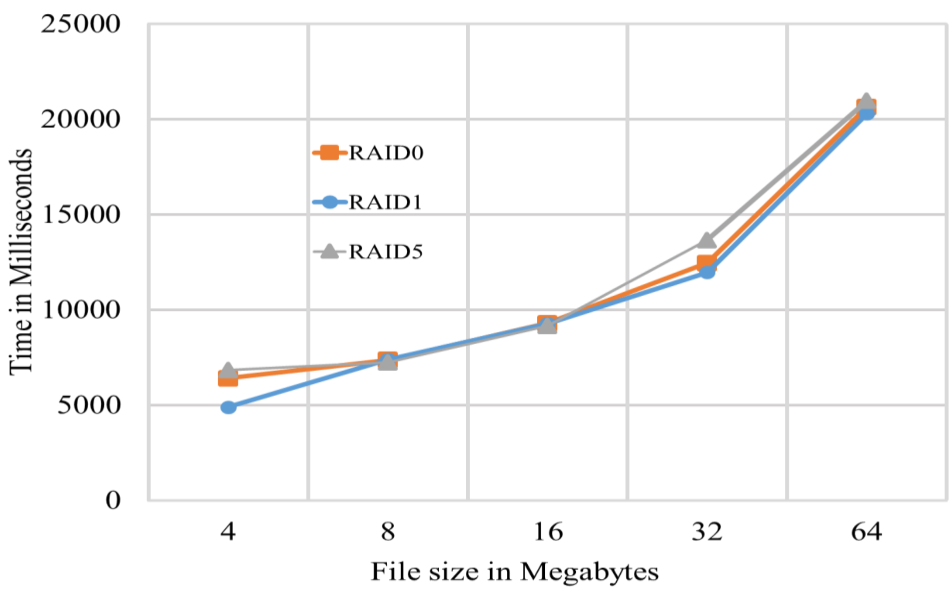


Figure 8. Flush Times for File Sizes between 4 to 64 MB

3.3.2 *Block Size*

RAID5 times are measures for write and read operations when sectors/block is varied from 128 to 1024 sectors. Figure 9 shows that while block size does not have much impact on read operations, larger block size gives better performance for write operations. For example, increasing the block size from 128 to 1024 (8 times), improves the write time by a factor of 1.68 (3895/2320). Read operations can thus use smaller block sizes while write operations can use larger block sizes. The disadvantage of using larger block sizes is that we observed more errors for write operations due to device driver limitations. The USB device architecture is limited to writing 40 sectors at a time as defined by its memory buffer. With larger block sizes, more iterations of 40 sectors will be done to complete the operation (until all sectors are done). Details of the underlying bare USB device driver are given in [24].

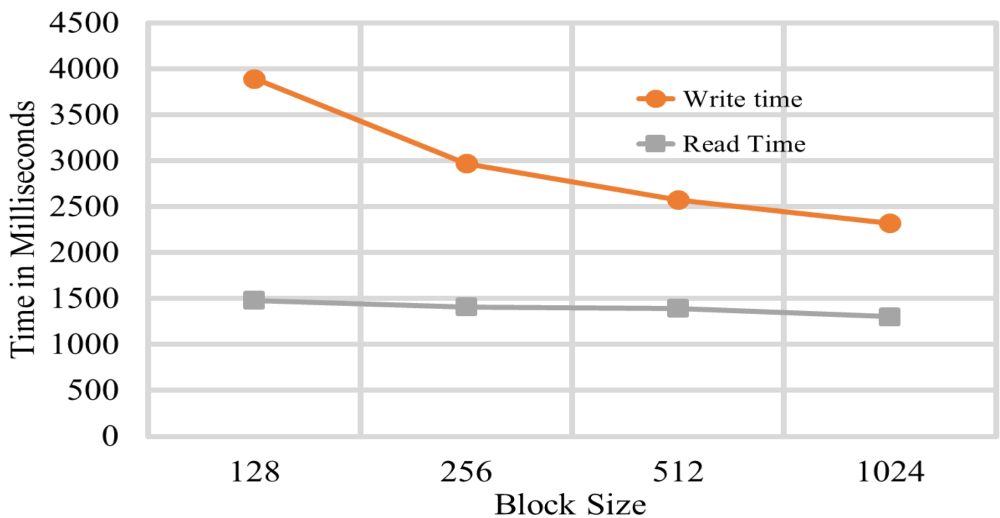


Figure 9. RAID5 Write and Read Times for 128 to 1024 Sectors/Block

The number of USBs varied for RAID0, and measured write and read times for a 64 MB file with 1024 sectors/block. Figure 10 shows that write times and read times converge as the number of USBs increase because there is a lesser load on each USB. This suggests that in a bare RAID file system using USBs, it is better to have more USBs to improve write performance (this may not apply to hard disks).

These measurements demonstrate the feasibility of implementing a bare RAID system and give some insight into designing similar systems to handle big data. It is also possible to build a distributed RAID system that uses only one large capacity USB per node.

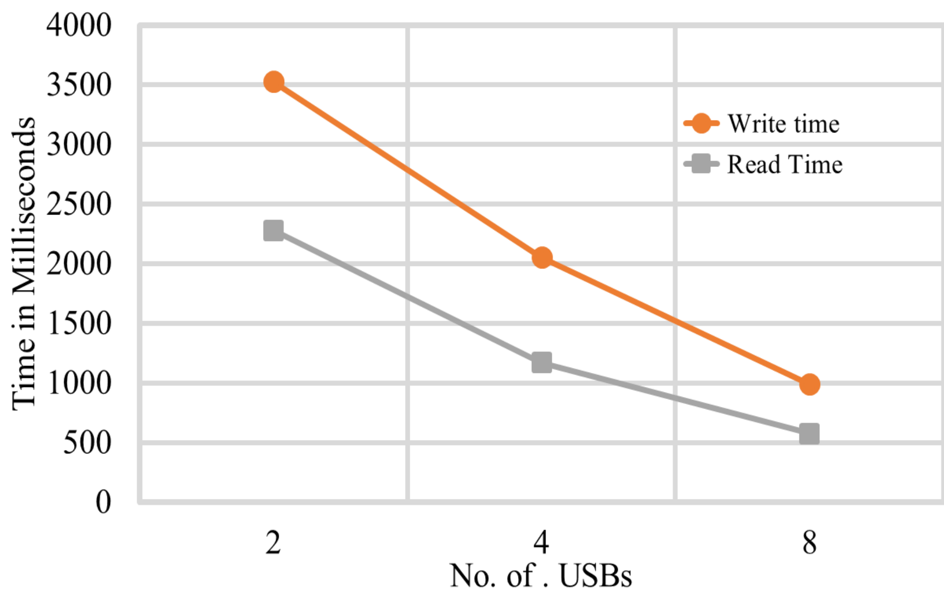


Figure 10. RAID0 Write and Read Times Using Multiple USBs (64 MB File, 1024 Sectors/Block)

3.4 Summary

This research described a new approach for designing RAID systems using bare machines and a USB-based file system. It also provided design and implementation details as well as preliminary performance data. Bare RAID systems limit the capability of an attacker who relies on OS or kernel facilities to compromise the system. They can thus provide a starting point for building secure OS-independent file systems, databases, cloud and big data storage systems, and novel mass storage applications. The feasibility of using bare machine RAID systems as an alternative approach for secure data preservation and long term archiving [38] without the need to reformat file data is also of interest. While the current bare RAID system implementation requires Intel x86 hardware, it could be in principle ported to other devices with different CPU architectures by redesigning the HAPI.

4 INTEGRATION OF SQLITE IN A BMC WEB SERVER

It is becoming increasingly common to hear that some social network data has been compromised or that some email or text messages have been stolen. Such security failures highlight the tradeoff between the need to keep sensitive information private and the convenience of having data accessible via public networks, where it can be compromised. Yet, most information systems that deal with private data or email communication are hosted on systems that are accessible via the Internet.

Since it is difficult for ordinary users to build communication systems that do not use the Internet or ISP services, a possible means to protect communication is to use end-to-end security together with smaller simpler clients and servers that are harder to attack. The BMC paradigm, which is inherently secure, provides an alternative to conventional OS-based systems for building such systems. It is especially suited for building secure applications that run on bare machines and enable select groups to communicate privately with no dependence on any external software such as an OS or OS libraries that are prone to exploitation. Such applications will be smaller and easier to secure than conventional applications that typically require some form of OS or kernel support.

In this section, we provide details of a BMC system that enables private communication among a limited group of users. The system provides database (DB) and novel DB mail services via SQLite for a limited group of users. It is based on the BMC paradigm that has inherent security by design.

4.1 System Overview

Figure 11 shows the architecture of the proposed system and the components of the integrated BMC Web server. The system is based on a bare PC Webserver [21] [20], which provides services to clients equipped with ordinary OS-based Web browsers. The clients can use PHP or HTML interfaces to access DB services, DB mail services, or Web services hosted on the BMC server. This multi-faceted system provides a baseline for secure private communication among individuals using standard Internet protocols and the Internet infrastructure. The popular SQLite DB engine is integrated with the BMC Web server to provide database facilities, but without using any OS or kernel. The DB client and DB mail client data are stored on a USB flash drive. SQLite's command line interfaces are used for system administration and debug purposes.

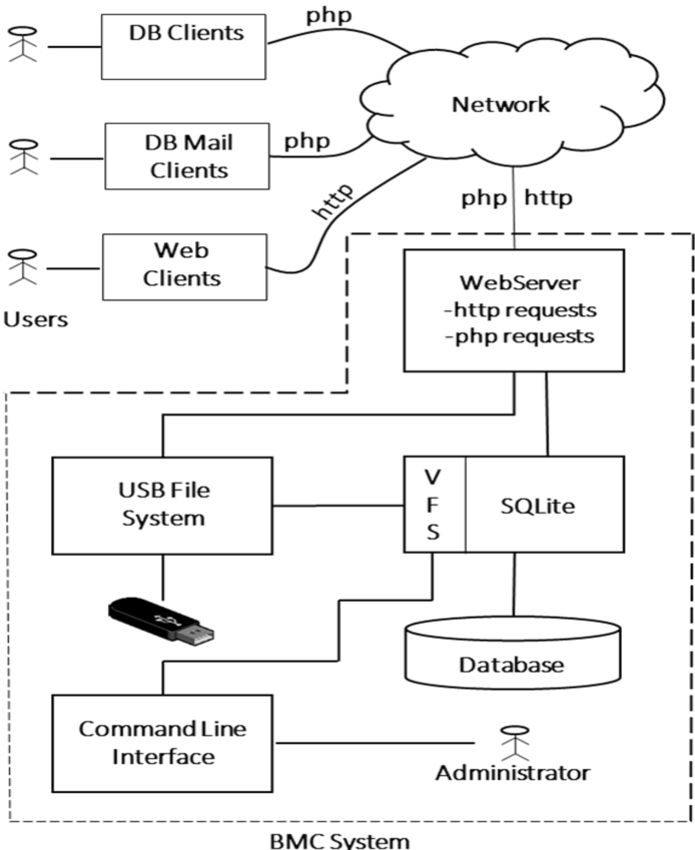


Figure 11. System Architecture

When the BMC server is booted, required files are transferred from an OS based (Windows) server using a lean version of TFTP (trivial FTP). These files are stored in memory and used during server operation. If there is a pre-existing database, it will also be loaded during this process. After the files are transferred, the BMC server runs to provide the necessary services to clients. A dummy DB needs to be created so that SQLite becomes operational (it needs a DB pointer to do any operation). Browser-based clients access pages stored on the BMC Web server using the HTTP protocol as usual, while DB clients can submit queries and access the SQLite DB database through a PHP interface. A “mail” table

is created to define the DB mail database. A "profile" for users with username and password is also loaded before the DB mail application starts.

The BMC server is a small simple system without OS dependencies that is easier to analyze for security vulnerabilities. It does not provide any facilities to run scripts, add new applications or load dynamic libraries. The system performs functions that are defined during development time, and allows new profiles to be loaded to add or delete users as needed. The system runs on any x86 architecture based PCs or Laptops.

4.2 System Integration

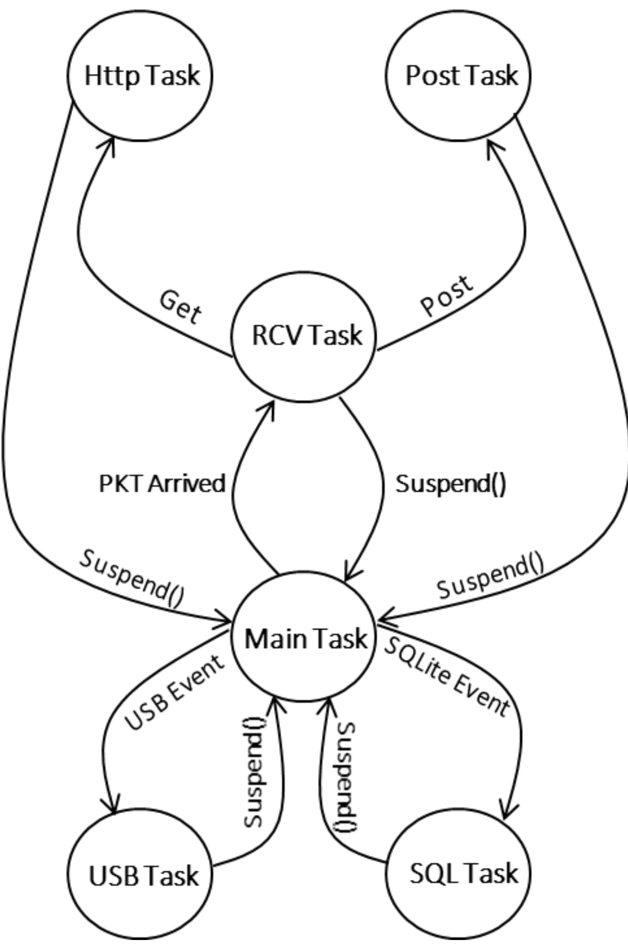


Figure 12. Task Interface

Many components need to be integrated to build this BMC server. In what follows, we give details of all components except for the bare PC USB file system, which is described in [39].

Tasks: There are six major tasks running in the system. The interaction between these tasks is shown in Fig. 12. When a system boots up from the USB, it presents a user menu to provide services. Using the "Load" option, the user can load the integrated server application. After loading, the "Run" option used to run the application, which starts the Main Task (MT).

There is only one MT in the system. In the MT, all other tasks are created, initialized and stored in their respective task pools. There is also a single Receive Task (RT) for receiving packets from the network. The four task pools in the system are: USB Task (UT), SQLite Task (ST), HTTP Task (HT) and Post Task (PT). These task pools reside in their respective stacks. Whenever a task is needed, it is popped from the stack and placed into a circular list for execution. When a task is complete it is pushed back onto the stack. The circular list maintains the current tasks and processes each in a first-in-first-out (FIFO) manner. When a packet arrives, the MT starts the RT. The RT runs as a single thread of execution without interruption until the packet is processed. If the packet has a GET request an HT is started, and if it has a POST request a PT is started. A USB event such as insert, remove, read or write will invoke a USB task to process the event. Similarly, a client's new SQLite event will start a SQLite task. For simplicity, this paper only considers one SQLite event (i.e., a single SQLite client); however, the system design and implementation does not limit the number of SQLite events (clients).

SQLite Integration: The SQLite code integrated in this system is taken from the amalgamated package [2], which has two "C" files: shell.c and sqlite3.c. The shell.c has "main()" and other user interfaces. When this code was transformed to run on a bare PC, all system call interfaces were replaced with bare PC interfaces [4].

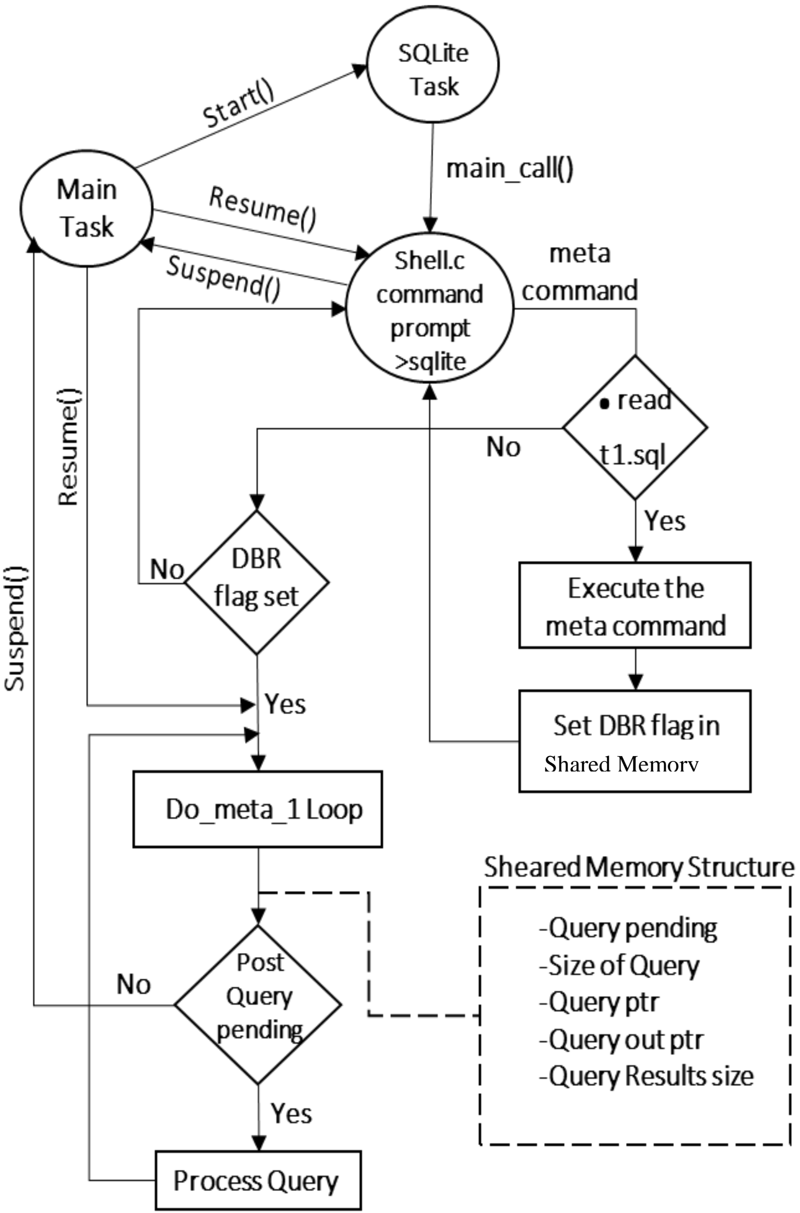


Figure 13. Integration of SQLite System

Fig. 13 shows the processing logic to integrate the bare PC SQLite system with the rest of the application. We changed the "main()" call in the shell.c file to "main_call()", so that

there is only one main() in the system. The MT starts ST, which in turn calls “main_call().” The ST runs along with the other tasks in the integrated Web server application. When ST starts, it prompts **“>sqlite”** and waits for the user requests. At the server side, all SQLite commands are run using this command line interface.

The original SQLite is designed as a client. To integrate SQLite in the bare PC Web server, we created a dummy database. In “shell.c” there is an instance of a database, and appropriate initializations are done in the file “sqlite3.c.” Otherwise, SQLite will not create any database instance. This approach allows us to use the SQLite code as is and simply tap into the database interface in the file “shell.c.” As shown in Fig. 13, a small script file t1.sql is used for this purpose. It includes the do-meta-command “.read t1.sql” to create a dummy database and a dummy table t1. We can enter other SQL statements or meta-commands at the command prompt before running the script file. Once the script file is run, the SQLite interface at the command line interface is disabled as the “.DBR” meta-command sets a DBR flag in shared memory. The “.DBR” meta-command is added to the SQLite set of commands.

When the DBR flag is set, the SQLite shell program will go into the “do_meta_1” loop. When necessary, ST suspends itself and waits for the client requests. The suspend() function returns to the MT. The suspend() function in C++ was mapped to the suspend() function in C (as class instantiation cannot be done from our C program). Once the DBR flag is set, SQLite processes client queries via a POST command.

Many issues were addressed when integrating SQLite database capability into the bare PC Web server. They involved the file, task and client interfaces to SQLite. Other

implementation issues involved communicating from the C code to the C++ code, which required special pointer conversions and prototypes.

4.3 Design Insights

Writing applications in a BMC environment is quite different than writing applications in a conventional system, which typically requires an OS platform or some form of a kernel. The BMC application methodology as described in [40] illustrates this concept. The BMC programming paradigm may be viewed as a holistic approach, where one needs to consider all aspects of application and system programming, and the execution environment. As there is no OS or centralized kernel running in the system only a user mode is supported, and the programmer writes BMC applications that consist of a sequence of events executing in an interleaved fashion. Each event code becomes a small thread of execution that runs without any interrupts or task switching. The programmer has the total control of both application and execution flow. For network communication, BMC applications have their own Ethernet driver and a complete lean TCP/IP stack that implements the protocols needed by the application.

A network event in the integrated Web server application occurs when a TCP packet arrives with a SYN flag. As shown in Fig. 14a and Fig. 14b, client and server go through several states when sending and receiving packets during the connection. In the integrated system, either an HTTP GET request or a HTTP POST request is sent by the client. When the client needs data from the server a GET request is used, and when it sends data to the server a POST request is used. For a GET request, the server knows how many packets to send and it can control the data transfer. For a POST request, the client controls the data transfer. The number of packets for a POST request depends on the content length sent in

the first packet. The server keeps track of the number of packets and the total data size to verify that all packets have arrived.

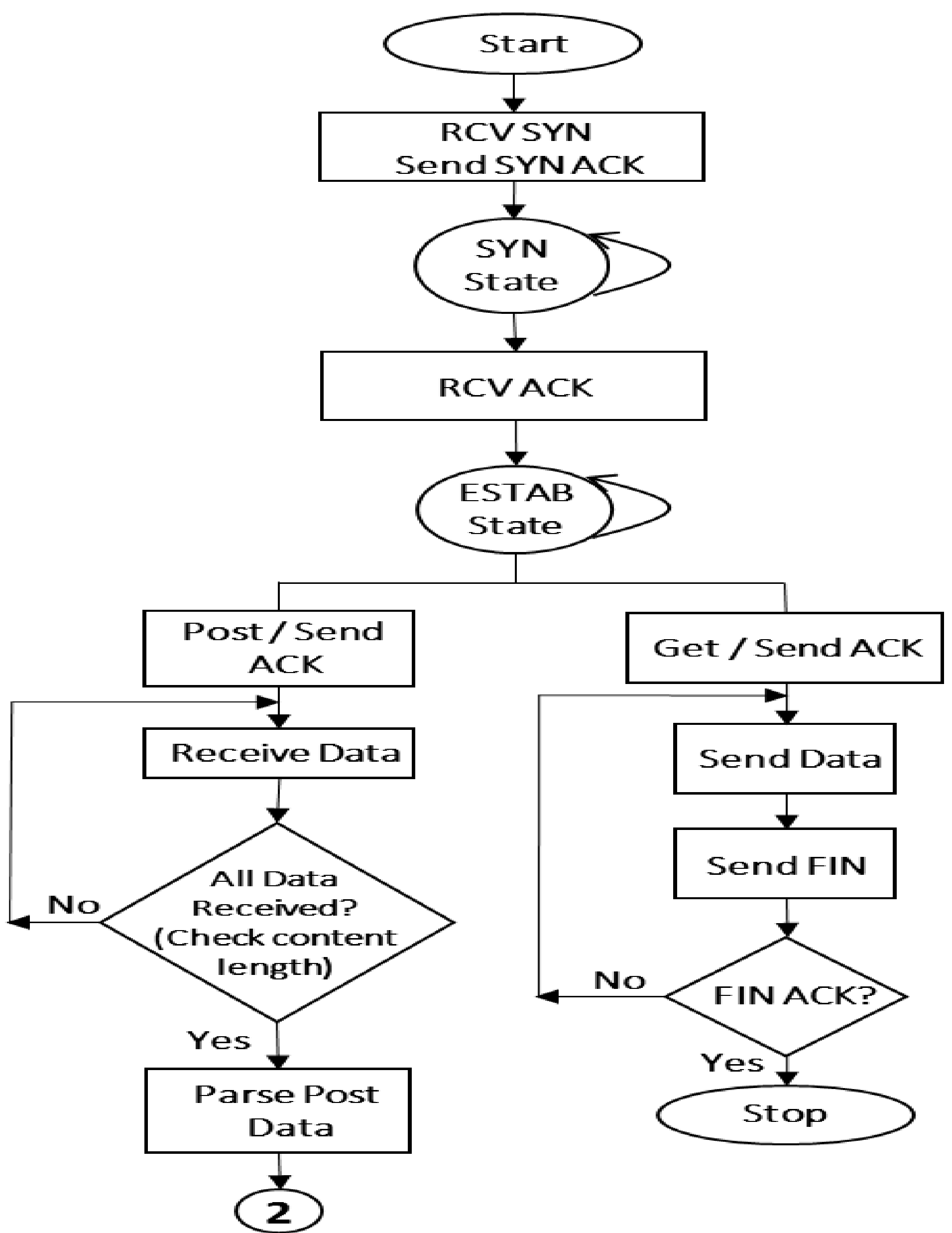


Figure 14a. Network Flow Control

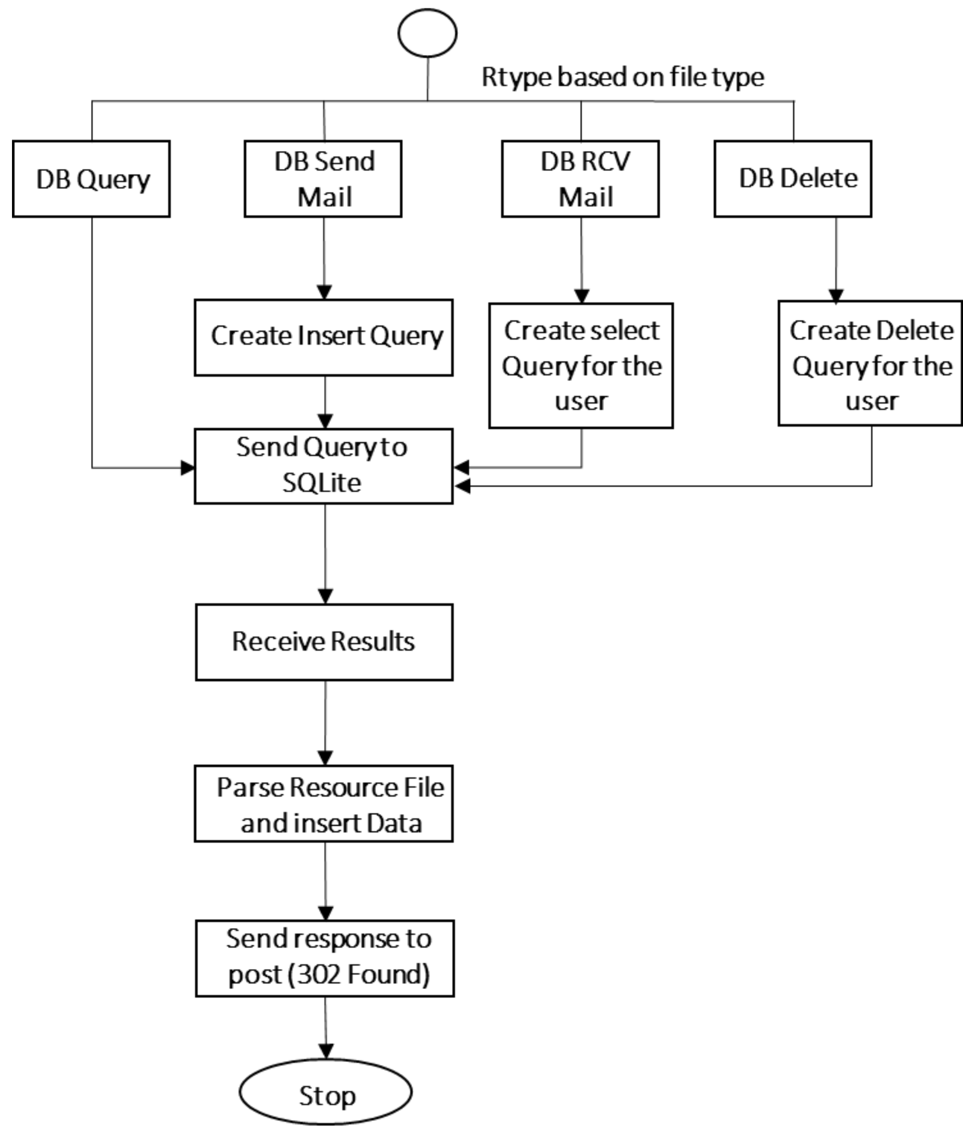


Figure 14b. Network Flow Control (cont.)

POST data processing requires a PHP parser. To avoid complexity only the functionality needed for the integrated application is implemented in our PHP parser. Since we are designing a server system, we are able to control the design of the PHP files and their contents.

It should be noted that HTML files are static content and PHP files are dynamic content. The database access from the client requires dynamic content due to the dynamic nature of queries and DB mail. When a POST arrives at the server, a packet of data at a time is received until all packets of data are received. Then the server will process the data according to the type of the request. There are five types of interfaces from the client to the server, which uses five forms. These types are shown in Table 3. Each type of form corresponds to a different type of PHP file. In addition, the client fetches the results from the DB using a separate form named "inbox.php." Thus a total of six PHP files are required in the system. Parsed attributes of each file are also shown in Table 3.

Function	File name	Operation
Login	Login php	Extract username, password
DB Query	Compose.php Inbox.php	Extract query Put results
Send Mail	Sendmail.php	Extract mail parameters Form Insert query
Receive mail	Retrieve.php Inbox.php	Extract username Form select query Insert results in file
Delete mail	Delete.php	Extract username Form Delete query

Table 3. PHP Files and their Parsed Attributes

When the "login.php" file comes in as POST data, the server parses this data for the username and password, validates them and logs the information. This information is used throughout the session for a given user while logged in. Only necessary information is

parsed using the keywords; the file also has keywords to help the parser find the key reference points.

Database queries from a client will come in as a "compose.php" file to the server through the POST command. The server extracts the query data from the packet(s) and forms a query block for the SQLite server, which returns the results in a memory block. These results are inserted into the "inbox.php" file at the insertion points shown in Fig. 15. The insertion points "POINT1" and "POINT2" provide the references in the files to insert data into the file. When the result data is inserted, the file size may increase and the "POINT2" reference will be moved down to increase the file size according the size of the data inserted in the file. This technique enables us to avoid writing a full PHP parser.

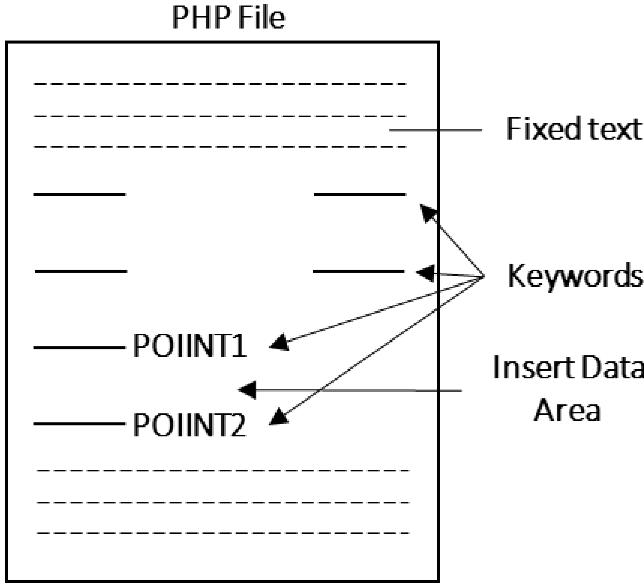


Figure 15. PHP File

For sending mail, the “sendmail.php” file is used to form a mail message. At the server, mail attributes such as from, to, subject, and body are extracted, and an INSERT query is formed. This query is sent to SQLite for execution and checked for a good return code. Parsing of the “sendmail.php” file is similar to the other PHP files. In this case, a username is also parsed from the mail content and validated by comparing with the login name.

Receiving mail is more complex than the other client requests. The “retrieve.php” file sent to the server through the POST command contains the username of the receiver, which is used to retrieve mail from all other users. A SELECT query is formed to retrieve emails for a given user from all other users (if any). This query is sent to SQLite and the results are obtained. These results are then inserted into the “inbox.php” file, and the client sends a GET request to see the results. Inserting results into the file “inbox.php” is similar to inserting database query results.

Finally, the “delete.php” file comes in to the server as POST data. At the server, the username is extracted from the message and the appropriate query is formed to delete the mail message from the database. For a given user, it deletes mail from a sender based on the sender’s username. Only one mail message at a time can be deleted.

The current version of the server is a prototype that demonstrates the feasibility of integrating SQLite and provides basic functionality to illustrate handling of DB queries and DB mail. It can be easily extended to perform more sophisticated functions as it is designed in a modular fashion.

4.4 Implementation

The general methodology to implement BMC applications is described in [40]. SQLite integration is implemented in C/C++ with direct hardware interfaces. The server

application uses an Intel Gigabit NIC (on-board chip) and its corresponding BMC driver. The executable size is 821,248 bytes. This includes the application code and the required execution environment code that makes it a self-contained, self-managed and self-controlled application. A user controlled USB drive contains this executable including its boot code, which is used to boot up a bare PC and run the server application. The application runs on any Intel x86 architecture based PC or laptop. The SQLite source code is 140K lines, and POST source code is 3K lines.

4.5 Functional Operation

This section describes the functional operation of the server when handling DB mail and DB queries. It also shows some results from running the server including example screens and pages displayed at the client, a bare server display, and a Wireshark network packet trace.

4.5.1 *Server Operation*

When a system is powered up, the Web server runs and the display shows the SQLite prompt in Fig. 16, and other information that is used for diagnostics and debugging. An administrator can also enter command line queries and get the response on the screen. Administrative functions such as loading user profiles and creating a dummy database can be done before clients access the database. Reading the “t1.sql” file makes the database ready for client operations. At this point the DBR flag is set in memory so that POST commands will be processed for client requests. The server screen shows that STand MT are running. As noted earlier, RT runs whenever a new packet arrives, a PT runs when a

POST command is received, an HT runs when a GET request is received, and a UT runs when a USB event occurs. The server will keep running until it is powered down.

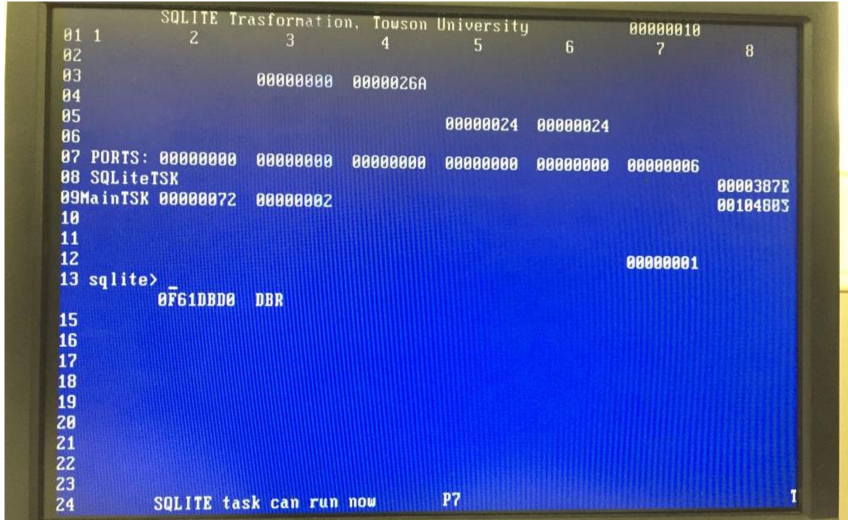


Figure 16. Server SQLite Interface

Fig. 17 shows a Wireshark trace with an HTTP GET request for the “barerocks530k.gif” file. Intermediate packets are deleted to minimize the trace for display.

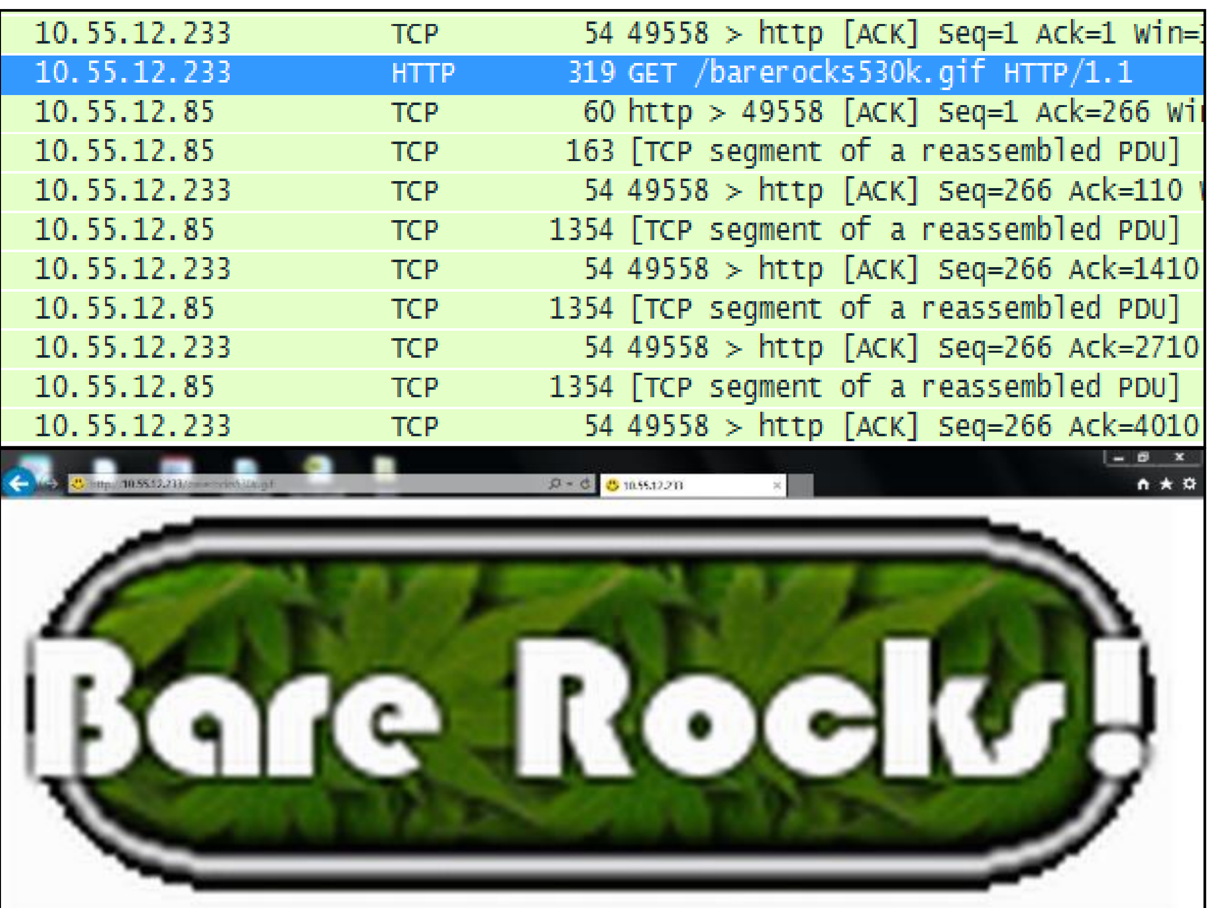


Figure 17. A Wireshark Trace for GET and The Client Page

Fig. 18 shows a Wireshark trace for read mail with a POST (which has the user name) and an HTTP GET request for the “inbox.php” file containing the results.

```
10.55.12.233    TCP     54 49610 > http [ACK] Seq=1 Ack=1 Win=17520 Len
10.55.12.233    HTTP   536 POST /b_retrieve.php HTTP/1.1  (application/
10.55.12.85     TCP     60 http > 49610 [ACK] Seq=1 Ack=483 Win=1500 Le
10.55.12.85     HTTP   324 HTTP/1.1 302 Found
10.55.12.233    TCP     54 49610 > http [ACK] Seq=483 Ack=271 Win=1725
10.55.12.233    HTTP   433 GET /inbox.php HTTP/1.1
10.55.12.85     TCP     60 http > 49610 [ACK] Seq=271 Ack=862 Win=1121
10.55.12.85     TCP    192 [TCP segment of a reassembled PDU]
10.55.12.233    TCP     54 49610 > http [ACK] Seq=862 Ack=409 Win=1711
```

Bare Machine Database

Read mail:

bpc0001 | Retrieve

Figure 18. A Wireshark Trace for Read and The Read Mail Page

4.5.2 *Database Application*

The SQLite DB application runs as a client server system. A client access the Web page shown in Fig. 19. In order to access the DB or DB mail, a user clicks on the DB link. Then the login interface in Fig. 20 appears. The user enters the username and password which are validated and the system menu is displayed as in Fig. 21. Selecting the DB Query option, a user can enter a SQLite query as shown in Fig. 22. The results of the query are displayed as in Fig. 23. Database queries can be typed in as text in the window or they can be included in an attachment. The system is tested for both options.

Figure 19. Database Access Page

A database application running on the system can be accessed by users and administrators. The bare PC Web server has a debug mode, where some Web pages can show the internal server operations and DB operations. DB queries are limited to the queries that can be performed by SQLite.

Bare Machine Database

Username:

bpc0001

Password:

Login

Figure 20. Login Page

Bare Machine Database

DB Query

DB Send mail

DB Receive mail

DB Delete mail

Logout

Figure 21. Menu Screen

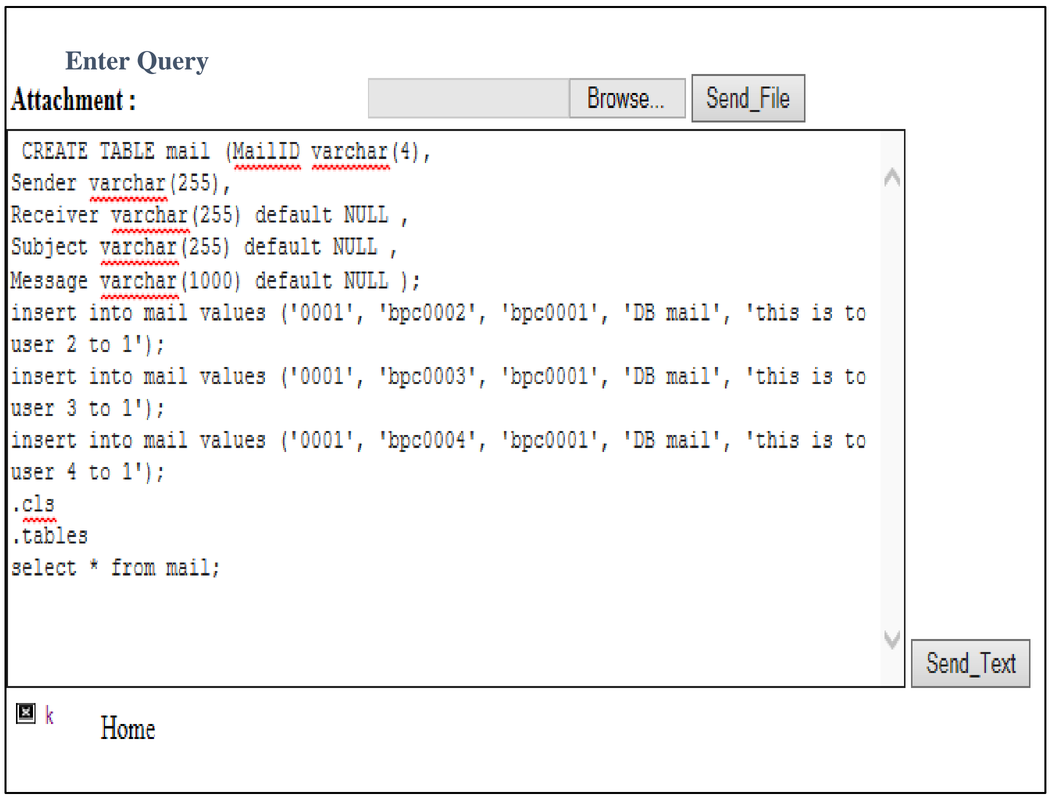
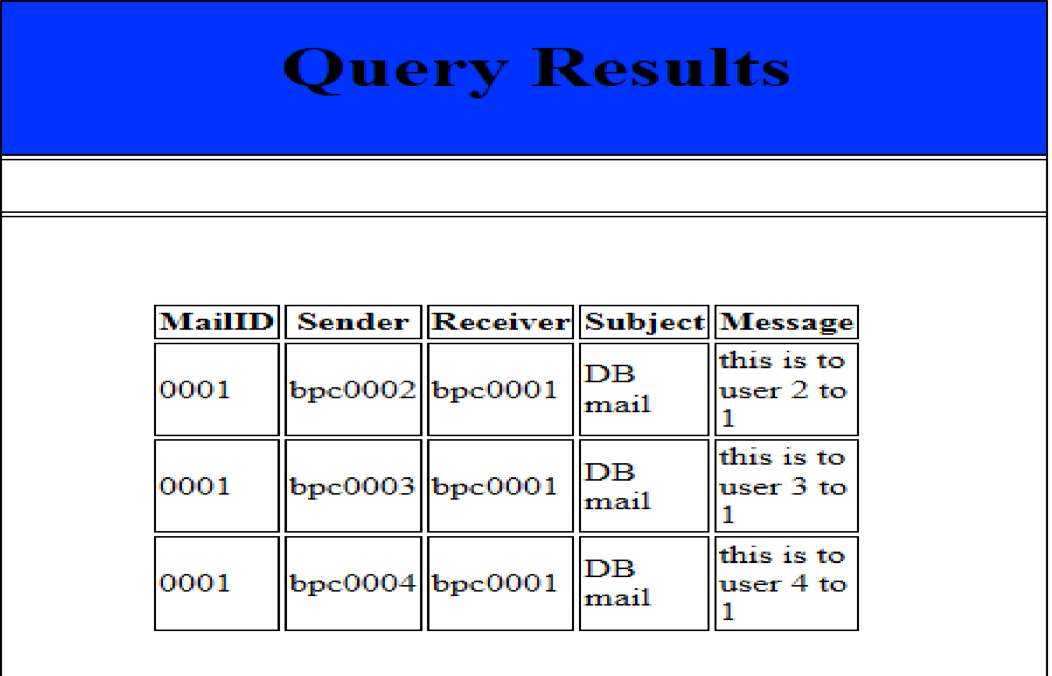


Figure 22. Query Page



MailID	Sender	Receiver	Subject	Message
0001	bpc0002	bpc0001	DB mail	this is to user 2 to 1
0001	bpc0003	bpc0001	DB mail	this is to user 3 to 1
0001	bpc0004	bpc0001	DB mail	this is to user 4 to 1

Figure 23. Query Results Page

4.5.3 *Database Mail Application*

The SQLite database mail application is referred to as DB mail. DB mail is different from other mail systems. It is designed to work with a limited group of users who share the SQLite database to send and receive email. They login to the Web server and send/receive email to/from their peers. The DB mail system does not include TLS, but it can be added by integrating with a TLS Web server [14]. The SQLite DB is created by a DB administrator in a physically secure manner. In the same way, user accounts and passwords are also given by the administrator. Any user can send email to any other user in the group. The clients can be any users with a Web browser on any standard OS platform. Email messages sent by a client are stored in the database. When a user is logged in, they can read their email or delete the email from a given user. DB mail does not allow any spam since it is only for a select group of users. More sophisticated functions can be added to the system to provide additional features as needed.

DB mail operations can be send, receive, or delete. A send mail form is shown in Fig. 24. It requires sender name, receiver name, subject, and message body. This form is sent to the server as a POST message. The server will insert this query into the mail database and return.

A receive mail form is shown in Fig. 25. It requires the username for the user who wants to receive mail. The results of receive mail are shown in Fig. 26. All emails received for this user are displayed. The server receives the username and password and forms a query. It submits the query to SQLite and collects results. The results are stored in a file and returned to the client through a GET request.

Similarly, the delete mail form is shown in Fig. 27. It requires the sender username for which a message is to be deleted. A user must be logged in (as a receiver) to delete sender messages.

Send

From : bpc0002

To : bpc0001

Subject : info

This is DB mail from user 2 to user 1.
Hello.

Figure 24. Send Mail Page

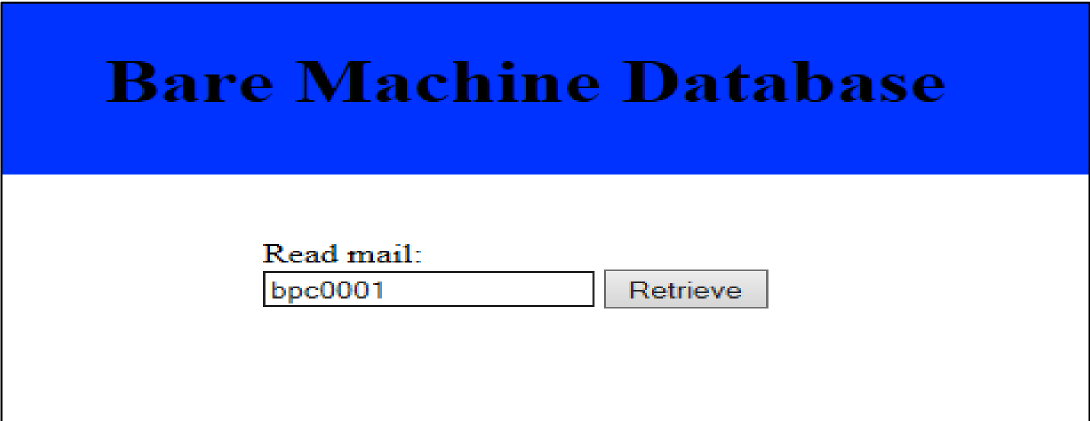
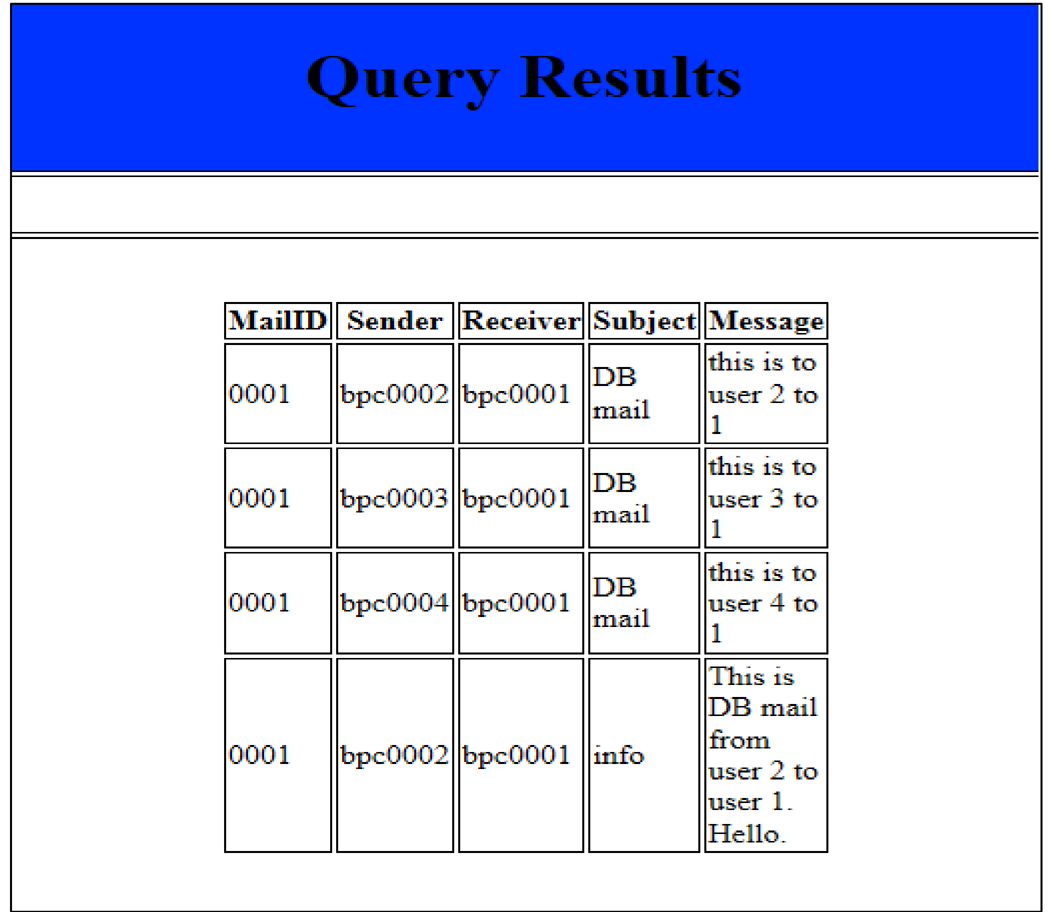


Figure 25. Receive Mail Page



MailID	Sender	Receiver	Subject	Message
0001	bpc0002	bpc0001	DB mail	this is to user 2 to 1
0001	bpc0003	bpc0001	DB mail	this is to user 3 to 1
0001	bpc0004	bpc0001	DB mail	this is to user 4 to 1
0001	bpc0002	bpc0001	info	This is DB mail from user 2 to user 1. Hello.

Figure 26. Receive Mail Results Page

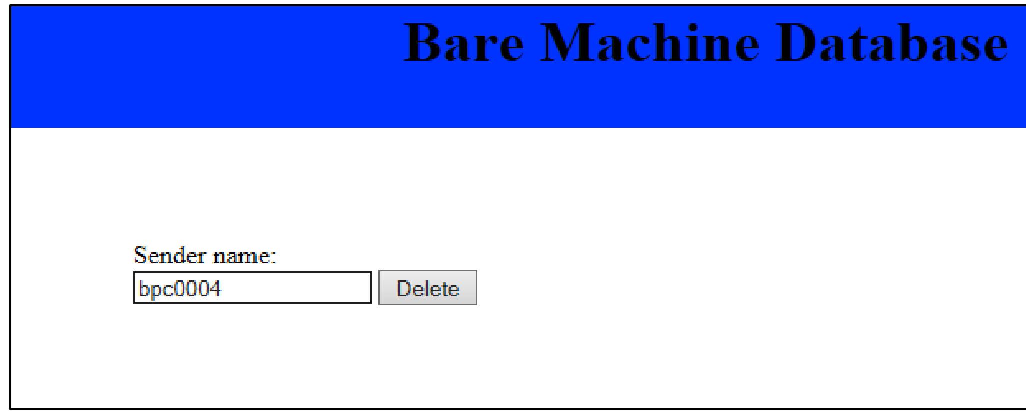


Figure 27. Delete Page

5 SIGNIFICANT CONTRIBUTIONS

This research serves as a foundation for integrating complex BMC applications. We showed how to run a BMC Web server, database and file system as a single monolithic executable providing a new model for storage systems. This integrated model is a building block to construct BMC systems with simplicity and security by design. Using this model, it becomes possible to integrate more complex databases such as MySQL and other database systems into Web-based systems. The email system with an integrated database enables a select group of users to communicate in a closed system environment with controlled security and privacy. The inherent security by design in this model provides an alternate way of managing storage using BMC paradigm. When storage becomes independent of the OS and environment, the stored data can be used for long term archiving that is not prone to obsolescence and porting as long as the underlying CPU architecture is compatible. It is not difficult to add a direct hardware API for a new instruction set architecture and still preserve the rest of the code in the application.

6 CONCLUSION

In this dissertation, we described the implementation of RAID on a bare PC file system. We also showed how to integrate an open source database system such as SQLite to work with existing bare PC applications. Our research shows how to use this system for real world client/server applications such as DB and DB mail. Finally, it enables SQLite to run in a client/server environment.

The integration of SQLite into a bare PC Web server illustrates a novel approach to integrate a variety of applications based on the BMC paradigm. The DB and DB mail applications in the BMC Web server are used by ordinary clients with OS-based Web browsers. Our design and implementation shows how to run the existing SQLite code as is in a bare PC environment instead of an OS platform.

The integrated system provides DB mail for a select group of users without the complexity and vulnerabilities of general purpose public email systems. It can be used by a closed group of users for secure private communication in military, government or commercial organizations. The ability to run SQLite in a bare PC environment shows that it is possible to run existing code in pervasive devices without changes. Furthermore, if applications are built using the BMC paradigm, they can be run ubiquitously on many CPU architectures.

Our research is a first step towards integrating other complex open source applications to run on a bare PC or a bare machine system. This approach avoids the use of a conventional OS or kernel eliminating both the security and performance drawbacks of conventional systems. Future research can build on this work by integrating other commercial database systems with BMC applications.

APPENDIX

In this appendix, we give internal details of the BMC systems we have implemented. In many cases, we provide relevant parts of the actual code.

A. Compilation Environment

The bare PC file system is written entirely in C/C++. The file system uses a newly developed USB device driver and existing bare PC C++ API calls to interface with the hardware. A bare PC C++ API call invokes a C call, and that in turn invokes an assembly call. The compiling environment uses batch files to compile and link the file system application with the necessary bare PC code. The Visual Studio C++ compiler (batch mode), MASM 6.11 assembler, and Turbo assembler were used to create executable modules. We have written batch files to do compilation and linking for boot and loader programs and the file system application.

B. Operating Environment

RAID: An Optiplex 960 PC was used for RAID implementation and for SQLite Web server integration. This PC has 4MB of memory and ten USB ports. It has two USB controllers, 4 and 6 ports in each. The most significant implementation for RAID involves roaming across USB ports to slice data across them. The "processRequest()" function contains complete code for processing a RAID request as shown in Figure 28-33. This code illustrates the details of RAID processing.

processRequest() function code

```
176 //*************************************************************************************
177 // process USB task request
178 //  2 sector read and 38 sector write in a loop works in state 0x95
179 //  test unit ready in a loop works in state 0x95
180 //  taskindex is portno - 1
181 //  tasknumber[] stores task ids, using taskindex as index
182 //*************************************************************************************
183 int USBFObj::processRequest()
184 {
185     AOATask task;
186     apptask tsk;
187     UsbObj usbo;
188     FileObj fobj;
189     RAIDobj raidj;
190     int start_time = 0;
191         unsigned long iDatasize;
192         unsigned long iData;
193     int k1 = 0;
194     int uindex=0;
195     int bindex=0;
196     int k3= 0;
197     int retcode = 0;
198     int runflag = 0;
199     int i = 0;
200     int j = 0;
201     int k = 0;
202     int i1;
203     int *m1;
204     char *ptr1;
205     char p1;
206     int i2;
207     int slen = 0;
208     int flag = 0;
209     int addr = 0;
210     int devaddress = 0;
211     int nosectors = 0;
212     int suspendDelay = 0;
213     unsigned int mask1 = 0;
214     char *dptrr;
215     char *dptrw;
216     int readaddr =0;
217     int writeaddr=0;
218     char *wptr1;
219     char lbar[4];
220     char lbaw[4];
```

Figure 28. Process Request Part 1

```
221        char lbarx[4] = {0x00, 0x00, 0x20, 0x20};  //0th sector number, always +1
222        int readaddress;
223        int writeaddress;
224        int v2 = 0;
225        int fileHandle = 0;
226        char of1;
227        char f1Name1[60] = "Multiple USB test.txt";
228        char f1Name2[30] = "Hamdan Reconstructed.txt";
229        unsigned long startAddress;
230        unsigned long startAddress2 = 0;
231        unsigned long filesize = 0;
232        unsigned long filestartaddress = 0;
233        unsigned long fsize100;
234        unsigned long *att100;
235        unsigned long l1, l3;
236        char *fileaddrptr;
237        char *fileaddrptr2;
238            int k2;
239        int bi2=0;
240        int k4;
241        int k5 = 0;
242        char c1;
243        char c2;
244        int d0 =0;
245            int totalsectors = 0; //total sectors in the file
246            int sectorsinusb = 0; //no of sectors in each usb
247        int totalblocks = 0;
248        int noblocksinusb = 0;
249        fileaddrptr = &c1;
250        fileaddrptr2 = &c2;
251        att100 = &l3;
252        fsize100 = 1024*1024*4; //1 MB
253        *att100 = 123;
254        unsigned int count = 2000;
255        char * o_startAddress = (char *)RAID_TEMP_RW_ADDR - ADDR_OFFSET;
256
257        suspendDelay = 100;
258        m1 = &i1;
259        ptr1 = &p1;
260        lbar[0] = 0x00;  //used for mass storage address
261        lbar[1] = 0x00;
262        lbar[2] = 0x1f;  //0x1fc8, data1.txt starting sector
263        lbar[3] = 0xd1;
264
265        lbaw[0] = 0x00;  //used for mass storage address
```

Figure 29. Process Request Part 2

```
266     lbaw[1] = 0x01;
267     lbaw[2] = 0x18;
268     lbaw[3] = 0x81; //0x2310000
269
270     // lbaw1[0] = 0x00;  //used for mass storage address
271     // lbaw1[1] = 0x01;
272     // lbaw1[2] = 0x28;
273     // lbaw1[3] = 0x81; //0x2510000
274
275     taskindex = 2; //initial taskindex hard coded RKK
276
277     usbo.statec[taskindex] = 0;
278     m1 = (int*) (USB_MEMORY_ADDR + taskindex * 0x00008000) - ADDR_OFFSET; //memory to store data structures
279
280     //save controller addresses
281     //---------------------------------------------------------------
282     retcode = io.AOAgetSharedMem(0x15C);  //get controller 1 address
283     io.AOAsetSharedMem(0x164, retcode);
284     retcode = io.AOAgetSharedMem(0x160);  //get controller 2 address
285     io.AOAsetSharedMem(0x168, retcode);
286     //---------------------------------------------------------------
287
288     runController(1);  //initializes the controller
289
290     //create one file on bootable USB, which is the main file for RAID
291     //--------------------------------------------------------------------
292     taskindex = 2;
293     startUSB(2);
294
295     retcode = fobj.InitObj(taskindex, tasknumber[taskindex]); //currenttask);
296
297
298     ackUSB(taskindex); //ack it so that you can go to another USB
299     //--------------------------------------------------------------------
300
301     runController(1);  //initializes the controller
302     taskindex = 2;  //start with the boot USB
303
304     raidj.init();  //initialize raid object
305
306     // The following loop will only do RAW read/write to USBs for RAID
307         //  while (io.Application_Status != 2)
308
309   fileHandle = fobj.getFile(raidj.fname, &iData, &iDatasize);
```

Figure 30. Process Request Part 3

```
311     k5 = iDatasize/(RAID_BLOCK_SIZE*SECTOR_SIZE*NUMBER_RAID_USBS);
312     //RAID_BLOCK_SIZE is number of sectors in each block
313     totalsectors = iDatasize/SECTOR_SIZE;   //total sectors
314     totalblocks  = totalsectors / RAID_BLOCK_SIZE;
315     sectorsinusb = totalsectors/NUMBER_RAID_USBS;
316     noblocksinusb = sectorsinusb/RAID_BLOCK_SIZE;
317
318     k3 = 2*totalblocks/NUMBER_RAID_USBS;  //each iteration writes n blocks
319
320     //io.AOAtraceDword(k3);
321     //RKK-HAMDAN each step in the loop writes NUMBER_RAID_USBS
322     for (k4=0; k4 <NUMBER_RAID_USBS; k4++)
323       {
324     //io.AOAtraceDword(0x11111111);
325     io.AOAprintHex(k4, Line11+140);
326           if (connectStatus[taskindex] == 1)
327               startUSB(taskindex);
328           //k4 is 0 - 7 for USB index
329           //k3 (block index) is 0 - 15 for 4MB file size and 64 block size
330           //k3 is 0 - 3 for 4MB file size and 256 block size
331            t1w = io.AOAgetTimer();
332           for (i=0; i< k3; i++)
333           {
334                 retcode = raidj.RWriteOp(taskindex, currentcontroller, k4, i);
335            // raidj.clearCacheStorage();
336           }
337            t1wsum =  (io.AOAgetTimer() - t1w);
338                    t1r = io.AOAgetTimer();
339           for (i=0; i< k3; i++)
340           {
341                 retcode = raidj.RReadOp(taskindex, currentcontroller, k4, i);
342           }
343            t1rsum =(io.AOAgetTimer() - t1r);
```

Figure 31. Process Request Part 4

```
        ackUSB(taskindex); //ack it so that you can go to another USB
        task.AOAsuspendUSBTask(suspendDelay, usbo.statec[taskindex], 0,
         tasknumber[taskindex], 0, usbo.portno[taskindex]);
        // logic for switching to next controller
        if(taskindex == 5 && currentcontroller == 2)
        {
            // finished two controller USBs, go back to controller 1
            currentcontroller = 1;
            runController(currentcontroller);
            taskindex = 2;

        //ackUSB(taskindex);
            //startUSB(2);
        }
        else if(taskindex == 5 && currentcontroller == 1)
        {

            // first one is done, switch to 2nd controller
            currentcontroller = 2;
            runController(currentcontroller);
            taskindex = 2;
        }
        else    {
            // go to next USB
            // within the same controller
            taskindex++;
        }

    } //end of main while in this function



                        currentcontroller = 1;
            runController(currentcontroller);
            taskindex = 2;
            ackUSB(taskindex);
                    startUSB(2);

            fileHandle = fobj.getFile(raidj.fname, &iData, &iDatasize);
                     //get file size and data pointer for the file
             //file handle is returned
```

Figure 32. Process Request Part 5

```
395                 io.AOAclearLine(15);
396                 io.AOAclearLine(16);
397                     t1f = io.AOAgetTimer();
398                 retcode = fobj.flushFile(fileHandle, taskindex,
399                                     tasknumber[taskindex]);
400                 t1fsum= (io.AOAgetTimer() - t1f);
401                 io.AOAprintHex(t1wsum/4, Line16+20);
402                 io.AOAprintHex(t1rsum/4, Line16+40);
403                 io.AOAprintHex(t1fsum/4, Line16+60);
404                 io.AOAPrintText("Flush File After RAID.", Line17+80);
405                                         io.Application_Status = 2;
406                             io.AOAExit();
407         return 0;
408
409
410 }; //end of process request
411
```

Figure 33. Process Request Part 6

SQLite: The same system used for RAID is also used for SQLite integration. However, the operating environment for SQLite is a client server system, where SQLite is integrated with the BMC Web server and the clients can run on any Windows machines. The code snippets in shell.c that were modified for integration are shown in Figs. 34-41. A new command DBR is shown in Fig. 34. An interface function do_meta1 between SQLite and post processing is implemented as shown in Fig. 35. The shell_exe function in shell.c was modified in several places to interface with the Webserver and for post processing. This function is shown in Figs. 35 through 41. The SQLite code adopted here is designed to run as a command line interface. The command line interface runs as a separate SQLite task along with the Webserver. In order to make the SQLite run as a client/server system, the command line interface has to be disabled. It is done through running a DBR command. This command will set a DBR flag in the system. When this flag is set, the do_meta1 function call is used from the post process to interact with SQLite and the SQLite task is idle.

Shell.c code snippets

```
1771     if( c=='c' && AOAstrncmp(azArg[0], "cls", n)==0  && nArg==1 ){
1772       clearScreen();
1773       printScreenLayout();
1774     }else
1775    if( c=='D' && AOAstrncmp(azArg[0], "DBR", n)==0  && nArg==1 ){
1776         AOAPrintHex((long)p, Line14+20);
1777         AOAPrintText("DBR", Line14+40);
1778         CsetSharedMem32(0x60,1);
1779         //RKK-HAMDAN
1780         //save the dbrun ponter
1781 
1782     }
```

Figure 34. Shell.c Part 1

```
1127 //RKK-HAMDAN
1128 // new function to call from post object
1129 //  called to do SQLite functions
1130 //---------------------------------------------------------
1131 int do_meta1 ()
1132 {
1133   int rc=0;
1134   int rc1=0;
1135   char **pzErrMsg;
1136   struct callback_data *p;
1137   int c=0;
1138   int nArg=0;
1139   int n=0;
1140   char *zSql=0;
1141   int i=0;
1142   int *appStatus=0;
1143   sqlite3_stmt *mptr;
1144 
1145   //p = (struct callback_data *) (0x23008030 - 0x00110000);
1146   p = (struct callback_data *) (CgetSharedMem32(0x54)); //get p address saved before
1147   n = CgetSharedMem32(0x188);  //size of the query
1148   zSql = (char*)(CgetSharedMem32(0x184)); //query in pointer
1149   mptr = (sqlite3_stmt *)zSql;
1150   *appStatus = CgetSharedMem32(0x194);
1151 
1152   /* Error msg written here */
1153 
1154  //if( nArg==0 ) return 0; /* no tokens, no error */
1155   n = strlen30(zSql);
1156   c = zSql[0];
1157 
1158   if( c=='.' && AOAstrncmp(zSql[0], "", n)==0){
1159           rc = do_meta_command(zSql, p, appStatus);
1160   }
1161   else {
1162 
1163    AOAPrintText("R", Line2+20);
1164      rc = shell_exec(p->db, mptr, shell_callback, p, pzErrMsg);
1165      AOAPrintText("R2", Line2+20);
1166   }
1167   AOAPrintHex(rc, Line21+40);
1168   AOAPrintText(zSql, Line21+60);
1169   return 0;
1170  }
```

Figure 35. Shell.c Part 2

```
1177  ** Execute a statement or set of statements.  Print
1178  ** any result rows/columns depending on the current mode
1179  ** set via the supplied callback.
1180  **
1181  ** This is very similar to SQLite's built-in sqlite3_exec()
1182  ** function except it takes a slightly different callback
1183  ** and callback data argument.
1184  */
1185  static int shell_exec(
1186    sqlite3 *db,                              /* An open database */
1187     char *zSql,                         /* SQL to be evaluated */
1188    int (*xCallback)(void*,int,char**,char**,int*),   /* Callback function */
1189                                              /* (not the same as sqlite3_exec) */
1190    struct callback_data *pArg,               /* Pointer to struct callback_data */
1191    char **pzErrMsg                           /* Error msg written here */
1192  ){
1193    sqlite3_stmt *pStmt = NULL;     /* Statement to execute. */
1194    int rc = SQLITE_OK;             /* Return Code */
1195     char *zLeftover;          /* Tail of unprocessed SQL */
1196
1197    if( pzErrMsg ){
1198      *pzErrMsg = NULL;
1199    }
1200
1201    while( zSql[0] && (SQLITE_OK == rc) ){
1202      rc = sqlite3_prepare_v2(db, zSql, -1, &pStmt, &zLeftover);
1203        //AOAPrintHex(db, Line22+40);
1204        AOAPrintHex(rc, Line2+20);
1205      if( SQLITE_OK != rc ){
1206        if( pzErrMsg ){
1207
1208          *pzErrMsg = save_err_msg(db);
1209        }
1210      }else{
1211        if( !pStmt ){
1212          /* this happens for a comment or white-space */
1213          zSql = zLeftover;
1214          while( AOAisspace(zSql[0]) ) zSql++;
1215          continue;
1216        }
```

Figure 36. Shell.c Part 3

```
1217
1218        /* save off the prepared statment handle and reset row count */
1219        if( pArg ){
1220          pArg->pStmt = pStmt;
1221          pArg->cnt = 0;
1222        }
1223
1224        /* echo the sql statement if echo on */
1225        if( pArg && pArg->echoOn ){
1226          const char *zStmtSql = sqlite3_sql(pStmt);
1227          fprintf(pArg->out, "%s\n", zStmtSql ? zStmtSql : zSql);
1228        }
1229
1230        /* perform the first step.  this will tell us if we
1231        ** have a result set or not and how wide it is.
1232        */
1233        //AOAPrintHex((long)pStmt, Line22+120);
1234         AOAPrintText("  ", Line23+140);
1235         //AOAPrintText("01", Line22+140);
1236             //call from here when post comes
1237         //  but problem,it does not return
1238             rc = sqlite3_step(pStmt);
1239
1240        AOAPrintHex(rc, Line2+40);
1241
1242        /* if we have a result set... */
1243        if( SQLITE_ROW == rc ){
1244          /* if we have a callback... */
1245          if( xCallback ){
1246            /* allocate space for col name ptr, value ptr, and type */
1247          int nCol = sqlite3_column_count(pStmt);
1248          void  *pData = sqlite3_malloc(3*nCol*sizeof(const char*) + 1);
1249            if( !pData ){
1250           AOAPrintText("A8", Line22+20);
1251              rc = SQLITE_NOMEM;
1252        AOAPrintHex(rc, Line2+60);
1253            }else{
1254              char **azCols = (char **)pData;      /* Names of result columns */
1255              char **azVals = &azCols[nCol];       /* Results */
1256              int *aiTypes = (int *)&azVals[nCol]; /* Result types */
1257              int i, j;
1258          int rsize = 0;
1259          char *rptr; //results ptr
1260          char *hptr;
1261          j = 0;
```

Figure 37. Shell.c Part 4

```
1262    i = 0;
1263        AOAassert(sizeof(int) <= sizeof(char *));
1264    rptr = (char *)(CgetSharedMem32(0x18c));
1265
1266        /* save off ptrs to column names */
1267    //RKK-HAMDAN QUERYRESULTS
1268
1269    if (CgetSharedMem32(0x180) == 1)
1270    {
1271    hptr = (char*) (AOAconvertIntToHexChars(nCol));
1272    for (i=0; i<4; i++) //4 characters
1273       rptr[i] = hptr[i];
1274    rptr = rptr+4;
1275    rsize = rsize + 4;
1276    //store no of columns value in the first 4 bytes
1277    //no need for space after that
1278    }
1279
1280        for(i=0; i<nCol; i++){
1281          azCols[i] = (char *)sqlite3_column_name(pStmt, i);
1282      //prints column labels
1283          AOAPrintText(azCols[i], Line15+6+20*i);
1284      if (CgetSharedMem32(0x180) == 1)
1285      {
1286          AOAStrCopy (rptr, azCols[i], AOAstrlen(azCols[i]));
1287      //column names are stored here
1288
1289      rptr = rptr + AOAstrlen (azCols[i]);
1290      *rptr = 0x99;
1291       rptr++;
1292      rsize = rsize + AOAstrlen(azCols[i])+1;
1293      }
1294      //AOAPrintText(rptr[i], Line12+6+20*i);
1295        }
1296        do{
1297          /* extract the data and data types */
1298      // printing results
1299          for(i=0; i<nCol; i++){
1300
1301            azVals[i] = (char *)sqlite3_column_text(pStmt, i);
1302        //rptr[i] = azVals[i];
1303        //AOAPrintText(rptr[i], Line11+6+20*i);
1304
```

Figure 38. Shell.c Part 5

```
1305            if (CgetSharedMem32(0x180) == 1)
1306            {
1307                AOAStrCopy (rptr, azVals[i], AOAStrlen (azVals[i]));
1308            //RKK-HAMDAN inserting row values
1309            rptr = rptr + AOAStrlen (azVals[i]);
1310            rsize = rsize + AOAStrlen(azVals[i]);
1311            rptr[0] = 0x99;
1312            rptr = rptr+ 1 ;
1313            rsize = rsize + 1 ;
1314            }
1315
1316          //prints results
1317                  AOAPrintText(azVals[i], Line16+6+i*20+j*160);
1318              //AOAPrintHex((long)&azVals[0], Line5+20);
1319
1320                  aiTypes[i] = sqlite3_column_type(pStmt, i);
1321
1322                  if( !azVals[i] && (aiTypes[i]!=SQLITE_NULL) ){
1323            {
1324                    rc = SQLITE_NOMEM;
1325        AOAPrintHex(rc, Line2+80);
1326            AOAPrintText("No memory", Line23+100);
1327            }
1328                    break; /* from for */
1329                  }
1330                } /* end for */
1331
1332                /* if data and types extracted successfully... */
1333                if( SQLITE_ROW == rc ){
1334
1335                  /* call the supplied callback with the result row data */
1336                  if( xCallback(pArg, nCol, azVals, azCols, aiTypes) ){
1337                    rc = SQLITE_ABORT;
1338                  }else{
1339          AOAPrintText("   ", Line23+140);
1340           AOAPrintText("02", Line22+140);
1341                    rc = sqlite3_step(pStmt);
1342        AOAPrintHex(rc, Line2+100);
1343
1344                  }
1345                }
1346                j++;
```

Figure 39. Shell.c Part6

```
1346            j++;
1347          } while( SQLITE_ROW == rc ); //end of do while
1348
1349      //-------------------------------------------
1350      //insert zzzz pattern and no of rows and
1351      //  total number of bytes for the output
1352      //  PHP file can read this values and parse
1353      //  the query results
1354
1355      if (CgetSharedMem32(0x180) == 1)
1356      {
1357      for (i=0; i<4; i++)
1358         rptr[i] = 'z';//zzzz pattern
1359      rptr = rptr + 4;
1360      rsize = rsize + 4;
1361      hptr = (char*) (AOAconvertIntToHexChars(j));
1362      for (i=0; i<4; i++)
1363         rptr[i] = hptr[i];
1364      rptr = rptr + 4;
1365      rsize = rsize + 4;
1366      rsize = rsize + 4;
1367      hptr = (char*) (AOAconvertIntToHexChars(rsize));
1368      //add no of rows into results
1369      for (i=0; i<4; i++)
1370         rptr[i] = hptr[i];
1371       //save the query results size
1372       CsetSharedMem32(0x190, rsize);
1373      }
1374      //-------------------------------------------
1375
1376          sqlite3_free(pData);
1377    AOAPrintHex(rc, Line2+140);
1378        }
1379      }else{
1380        do{
1381     AOAPrintText("  ", Line23+140);
1382      AOAPrintText("03", Line22+140);
1383          rc = sqlite3_step(pStmt);
1384    AOAPrintHex(rc, Line12+20);
1385        } while( rc == SQLITE_ROW );
1386      }
1387    }
```

Figure 40. Shell.c Part7

```
1388        /* print usage stats if stats on */
1389        if( pArg && pArg->statsOn ){
1390          display_stats(db, pArg, 0);
1391        }
1392
1393        /* Finalize the statement just executed. If this fails, save a
1394        ** copy of the error message. Otherwise, set zSql to point to the
1395        ** next statement to execute. */
1396
1397        rc = sqlite3_finalize(pStmt);
1398        AOAPrintHex(rc, Line12+40);
1399        if( rc==SQLITE_OK ){
1400          zSql = zLeftover;
1401          while( AOAisspace(zSql[0]) ) zSql++;
1402        }else if( pzErrMsg ){
1403          *pzErrMsg = save_err_msg(db);
1404        }
1405        /* clear saved stmt handle */
1406        if( pArg ){
1407          pArg->pStmt = NULL;
1408        }
1409      }
1410    } /* end while */
1411
1412    AOAPrintHex(rc, Line19+40);
1413    return rc;
1414  }
1415
```

Figure 41. Shell.c Part8

C. Directory Structure

Figure 42 shows the directory structure used to build the applications described in this dissertation. Each directory in the list describes a class for that particular function. The /bin directory contains the Visual Studio compiler that is used to compile and link the programs using batch files. This directory structure is independent of any Visual Studio environment running on an OS platform. Each directory has a batch file that helps to compile assembly or C/C++ code. The /Webserver directory has the main program test.cpp to create executable test.exe. A single executable file is used that consists of all objects and runs as a single monolithic executable. There is an mk.bat file that creates a bootable USB and installs the test.exe along with other files onto a USB flash drive. This flash drive is the source of booting, loading and serving as a mass storage.

ARP	3/30/2018 3:04 PM	File folder
bin	3/30/2018 3:05 PM	File folder
dosclib	3/30/2018 3:05 PM	File folder
ethernet	3/30/2018 3:05 PM	File folder
ethernet3com	3/30/2018 3:05 PM	File folder
file	3/30/2018 3:05 PM	File folder
FileServer	3/30/2018 3:05 PM	File folder
fmtexe	3/30/2018 3:06 PM	File folder
FTOP	3/30/2018 3:06 PM	File folder
ftp	3/30/2018 3:06 PM	File folder
HTTPINDEX	3/30/2018 3:06 PM	File folder
httpParser	3/30/2018 3:06 PM	File folder
httptable	3/30/2018 3:06 PM	File folder
include	3/30/2018 3:06 PM	File folder
IncludeWin	3/30/2018 3:07 PM	File folder
INTERFACES	3/30/2018 3:07 PM	File folder
IP	3/30/2018 3:07 PM	File folder
lib	3/30/2018 3:08 PM	File folder
memorymap	3/30/2018 3:08 PM	File folder
nasm	3/30/2018 3:08 PM	File folder
post	3/30/2018 6:11 PM	File folder
sqlite	3/30/2018 3:08 PM	File folder
tcp	3/30/2018 3:08 PM	File folder
udp	3/30/2018 3:08 PM	File folder
usb	3/30/2018 3:08 PM	File folder
usbbootauto	3/30/2018 3:08 PM	File folder
usbfile	3/30/2018 3:09 PM	File folder
webserver	3/30/2018 3:09 PM	File folder
wwwt	3/30/2018 3:09 PM	File folder

Figure 42. Directory

D. USB Map

Figure 43 shows a USB map for storing files and other related information. Each USB has the same map and is stored in memory in different memory areas.

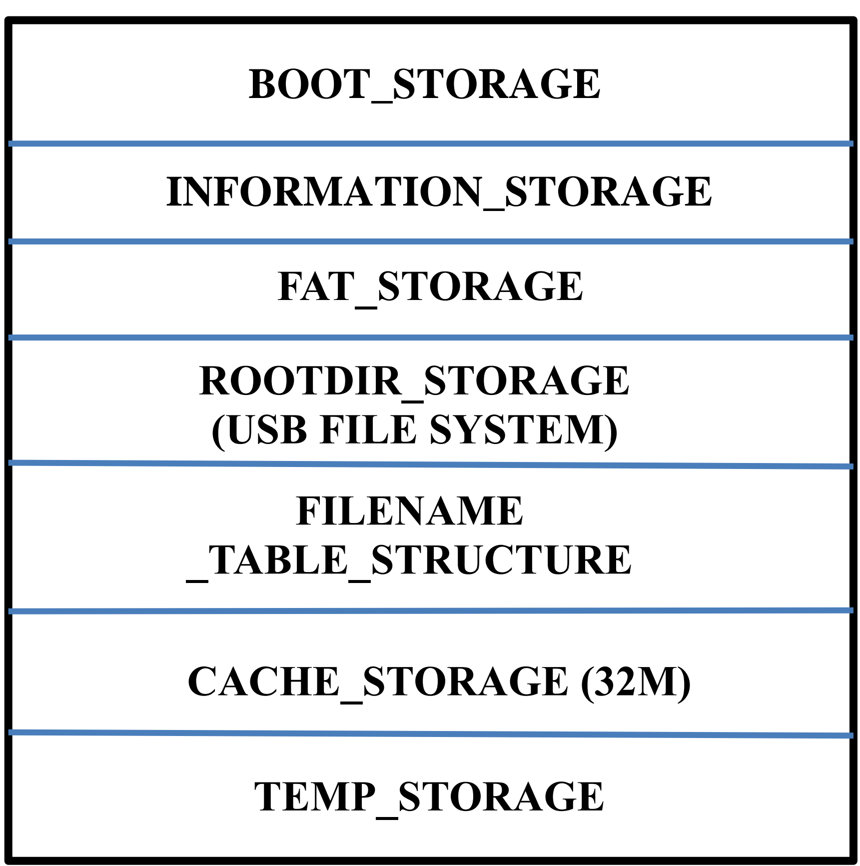


Figure 43. USB Map

E. Memory Map

Figs. 44 and 45 illustrate a memory map used for implementing applications as discussed in this dissertation. The test unit has 4GB of memory and the applications use only real memory without paging or virtual memory. For SQLite memory, there is a memory class object that provides an interface to malloc() called by SQLite program. A memory map is drawn for a given application suite and the designer makes the decisions for this map locations as needed. There is no actual memory management in the BMC applications.

Address	Region
00000600	PSP

00003900	Prcycle.exe
00111000	827,392 Bytes Test.exe 0xCA800
001DB800	--------------------
02110000	TSS_ADDR
0230A600	DPD_ADDR
0243A600	UDP_ADDR
0256A600	DPD_DATA
0389C600	UDP_DATA
0378C600	TASK_FUNCATION_PTRS
02E7A600	INT_PTRS
039ACA00	TCP_ADDR
04267840	TCP_MSG_ADDR
06A87480	WSTACK_ADDR
06B9F540	WCIRLIST_ADDR
07800000	TASK_STACK1
07808000	TASK_STACK2
0780D000	TASK_STACK8
08000000	FILE_ADDR
09000000	FILE_ADDR_START
0A000000	TRAGE1
0B000000	TRAGE2
0B800000	TASKLIST_TRACE_ADDR
0C000000	WTTRACE_ADDR
	STACK_ADDR
10000000	

Figure 44. Memory Map Part 1

Address	Region
11000000	INTERSERVER_PKT_ADDR
13130000	ARGV_ADDR
13230000	PT_BASE_ADDR
13238000	USB_BUFFER_ADDR
13538000	QH_ISTART_ADDR
13730000	QH_START_ADDR
13830000	QH_READDATA_ADDR
13930000	QH_WRITDATA_ADDR
13C00000	USB_MSGREORD_BASE
14800000	USB_MEMORY_ADDR
15800000	BOOT_STORAGE_ADDR
16401000	IS_STORAGE_ADDR
16402800	ROOTDIR_STORAGE_ADDR
16407000	ROOTDIRT_STORAGE_ADDR
16408000	FAT_STORAGE_ADDR
16600000	CACHE0_STORAGE_ADDR
18800000	FILENAME_TABLE_ADDR
18900000	DMAP_CACHE_DIR_ADDR
18908000	TRACE_STORAGE_ADDR
19000000	TEMP_STORAGE_ADDR
19008000	RTSTATUS_STORAGE_ADDR
19010000	RW_STORAGE_ADDR

Address	Region
1A000000	DB_MEMORY_ADDR
1A008000	DB_DATA_ADDR
1B000000	WDBPOST_ADDRI 16M
1C000000	WPACKETLIST_ADDR 16M
1D000000	PTEMP_STORAGE_ADDR
1E000000	MEM_BASE “SQLITE”
23000000	MEM_TBASE”SQLITE”
23000200	MEM_DBASE “SQLITE”
23008000	--------------------
28000000	FILE_STORAGE_ADDR
30000000	CHCHE_STORAGE_ADDR
32000000	TCP_POST_ADDR

Figure 45. Memory Map Part 2

F. PHP Files

Table 3 shows the PHP files used in the application. The files b_login.php, inbox.php, b_compose.php and sendmail.php are parsed by the application. Building a comprehensive PHP parser is expensive and is not needed for this application suite. To make the parser simpler, some keywords were placed in a PHP file. For example, b_login has two keywords b_uname and b_pword. These keywords are unique in the file. The PHP files are shown in Figure 46-56 for reference. The keywords are underlined in the figures to identify the keywords. In the inbox.php, two keywords POINT100 and POINT101, indicate the location where the query results to be inserted.

```
1  <?php
2      // if form not yet submitted
3      // display form
4      if (!isset($_POST['login']))
5      {
6  ?>
7  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
8  <html phpns="http://www.w3.org/1999/xhtml">
9  <head>
10 <meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1" />
11 <title>Bare DataBase Login</title>
12 <style type="text/css">
13 </style>
14 </head>
15 <body>
16 <table width="103%" border="0">
17   <tr>
18     <td height="107" bgcolor="#0033FF"><div align="center"><span class="style3">Bare Machine Database </span></div></td>
19   </tr>
20 </table>
21 <p><a href="index.html">Home</a>
22 <p>
23 <div id="Layer1">
24     <form action="" method="post" name="loginform" id="loginform">
25     Username: <br />
26     <input type="text" name="b_uname" />
27     <p>
28     Password: <br />
29     <input name="b_pword" type="password" />
30     <p>
31     <input name="login" type="submit" value="Login" />
32     <a href="index.html"></a>
33     </form>
34     <p>
35 <?php
36     //if form submitted
37     //check supplied login information
38     } else
39     {
40     $username = $_POST['b_uname'];
41     $password = $_POST['b_pword'];
42     //used to check if user entered information
43     if (empty($username))
44     {
45     die('Error : please enter your username');
46     }
```

Figure 46. b-login.php Part1

```
36        //if form submitted
37        //check supplied login information
38        } else
39        {
40        $username = $_POST['b_uname'];
41        $password = $_POST['b_pword'];
42        //used to check if user entered information
43        if (empty($username))
44        {
45        die('Error : please enter your username');
46        }
47        if (empty($password))
48        {
49        die('Error : please enter your password');
50        }
51        session_start();
52        $_SESSION['b_username'] = $username;
53        session_write_close();
54        header('Location: b_mail.php');
55        }
56    ?>
57    </div>
58    <p> </p>
59    <p> </p>
60    <p align="center">  </p>
61    <p> </p>
62    </p>
63    <p> </p>
64    <p> </p>
65    <p> </p>
66    </body>
67    </html>
```

Figure 47. b_login.php Part 2

```
6  <html xmlns="http://www.w3.org/1999/xhtml">
7  <head>
8  <meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1" />
9  <title>Query</title>
10 <style type="text/css">
11 <!--
12 #Layer1 {
13     position:absolute;
14     width:95px;
15     height:23px;
16     z-index:1;
17     left: -220px;
18     top: -6px;
19 }
20 #Layer2 {
21     position:absolute;
22     width:80px;
23     height:21px;
24     z-index:2;
25     left: 237px;
26     top: 112px;
27 }
28 #Layer3 {
29     position:absolute;
30     width:93px;
31     height:21px;
32     z-index:3;
33     left: 427px;
34     top: 112px;
35 }
36 .style3 {
37     font-size: 36px;
38     font-weight: bold;
39 }
40 .style4 {color: #CC3300}
41 -->
42 th, td {
43 border: 1px solid black;
44 }
45 </style>
46
47 <table width="100%" border="0">
48     <tr>
49       <td height="107" bgcolor="#0033FF"><div align="center"><span class="style3"> Query Results </span></div></td>
50     </tr>
```

Figure 47. Inbox.php Part 1

```
49      <td height="107" bgcolor="#0033FF"><div align="center"><span class="style3"> Query Results </span></div></td>
50    </tr>
51  <tr>
52      <td height="5"><a href="b_compose.php" target="_parent"><img src="http://images.google.com/url?source=imgres&amp;ct=tbn&amp;q=http
53  <form action="inbox.php" method="get" name="inbox" enctype="text/plain"  id="inbox" >
54
55    <td height="281" colspan="2"><label>
56        <p>b_NoOfColumns:         </p>
57        <p>b_NoOfTuples:          </p>
58        <p>DBTABLES: </p>
59        <p>          "b_msgbody"                    </p>
60        <table style="width:  20%">
61  <!--POINT100-->
62
63
64
65
66  <!--POINT101-->
67  </table>
68
69   </tr>
70   </label>
71   </td>
72   </tr>
73   </form>
74
75  </body>
76  </html>
```

Figure 48. Inbox.php Part 2

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1" />
<title>compose</title><style type="text/css">
<!--
.style1 {color: #FC324E}
#Layer1 {
    position:absolute;
    width:66px;
    height:25px;
    z-index:1;
    left: 394px;
    top: 252px;
}
#Layer2 {
    position:absolute;
    width:98px;
    height:21px;
    z-index:2;
    left: 11px;
    top: 137px;
}
.style3 {
    font-size: 36px;
    font-weight: bold;
}
-->
</style></head>

<?php
    // if form not yet submitted
    // display form
    if (!IsSet($_POST["send"])){
?>
<body>
<table width="103%" border="0">
  <tr>
    <td height="107" bgcolor="#FC324E"><div align="center"><span class="style3">DB Mail  </span></div></td>
  </tr>
</table>


<form action="" method="post" name=""  enctype="multipart/form-data" id="" >
<table width="90%" height="109%" border="0">
 <tr>
```

Figure 49. Sendmail.php Part1

```
47        <td height="24" colspan="2" bgcolor="#FFFFFF"><div align="center">
48          <div id="Layer1"></div>
49          <div align="left">
50            <div id="Layer2"><strong><span class="style1">       </span></strong></div>
51          </div>
52        </div>      </td>
53      </tr>
54      <tr>
55        <td width="69" height="24"><strong>From : </strong></td>
56        <td width="600"><label>
57          <input name="M_From" type="text" size="50" maxlength="1000" />
58           </label></td>
59      </tr>
60      <tr>
61        <td height="23"><strong>To : </strong></td>
62        <td><label>
63          <input name="M_To" type="text" size="50" maxlength="1000" />
64        </label></td>
65      </tr>
66      <tr>
67        <td height="10"><strong>Subject : </strong></td>
68        <td width="600"><label>
69          <input name="M_Subject" type="text" size="50" maxlength="1000" />
70        </label></td>
71      </tr>
72      <tr>
73
74          <input name="send" type="submit" id="Send" value="  Send  " />
75     </td>
76      </tr>
77      <tr>
78        <td height="281" colspan="2"><label>
79          <textarea name="M_Msgbody" cols="78" rows="17"></textarea>
80        </label></td>
81      </tr>
82  </table>
83  </form>
84  <?php
85      //if form submitted
86      //check supplied login information
87      } else {
88      $from = $_POST['b_from'];
89      $to = $_POST['b_to'];
90      $subject = $_POST['b_subject'];
91      $msgbody = $_POST['b_msgbody'];
92      //used to check if user entered information
```

Figure 50. Sendmail.php Part 2

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1" />
<title>Query</title><style type="text/css">
<!--
.style1 {color: #0000CC}
#Layer1 {
    position:absolute;
    width:66px;
    height:25px;
    z-index:1;
    left: 394px;
    top: 252px;
}
#Layer2 {
    position:absolute;
    width:98px;
    height:21px;
    z-index:2;
    left: 11px;
    top: 137px;
}
.style3 {
    font-size: 36px;
    font-weight: bold;
}
-->
</style></head>

<?php
    // if form not yet submitted
    // display form
    if (!IsSet($_POST["send"])){
?>
<body>
<table width="103%" border="0">
  <tr>
    <td height="107" bgcolor="#0033FF"><div align="center"><span class="style3">Bare Machine Database </span></div></td>
  </tr>
</table>


<form action="" method="post" name="compose"  enctype= "multipart/form-data"  id="compose" >
```

Figure 51. b_compose.php Part 1

```
46 <tr>
47   <td height="24" colspan="2" bgcolor="#FFFFFF"><div align="center">
48     <div id="Layer1"></div>
49     <div align="left">
50       <div id="Layer2"><strong><span class="style1">Enter Qurey </span></strong></div>
51     </div>
52   </div>     </td>
53  </tr>
54
55
56  <tr>
57  </tr>
58  <tr>
59    <td height="11"><strong>Attachment : </strong></td>
60    <td><label>
61
62      </label>    <input type="file" name="b_file" />
63          <input name="send" type="submit" id="Send" value="Send File"/>
64
65  </td>
66 </table>
67 </form>
68
69   <tr>
70     <td height="281" colspan="2"><label>
71 <form action="" method="post" name="compose"  enctype= "text/plain"  id="compose" >
72       <textarea name ="b_msgbody" cols="78" rows="17" type ="submit" > </textarea>
73             <input name="send" type="submit" id="Send" value="Send Text"/>
74       </form>
75     </label></td>
76   </tr>
77
78 </table>
79 </form>
80   <tr>
81     <td height="35"><a href="b_menu.php" target="_parent"><img src="http://images.google.com/url?source=imgres&amp;ct=tbn&amp;q=http://www.hd1080stockfootage.com,
82     <td width="9%">Home</td>
83   </tr>
84 <?php
85   //if form submitted
86   //check supplied login information
87   } else {
88   $from = $_POST['b_from'];
89   $to = $_POST['b_to'];
90   $subject = $_POST['b_subject'];
```

Figure 52. b_compose.php Part 2

```
83        </tr>
84    <?php
85        //if form submitted
86        //check supplied login information
87        } else {
88        $from = $_POST['b_from'];
89        $to = $_POST['b_to'];
90        $subject = $_POST['b_subject'];
91        $msgbody = $_POST['b_msgbody'];
92        //used to check if user entered information
93        if (empty($from)) {
94        die('Error : please enter mail from');
95        }
96        if (empty($to)) {
97        die('Error : please enter mail to');
98        }
99        header('Location: b_menu.php');
100       }
101   ?>
102   <p> </p>
103   </body>
104   </html>
```

Figure 53. b_compose.php Part 3

7 References

[1] " Bare Machine Computing.," Wikipedia, 2018. [Online]. Available: https://en.wikipedia.org/wiki/Bare_machine_computing.

[2] "The SQLite amalgamation.," SQLite, [Online]. Available: https://www.sqlite.org/amalgamation.html. [Accessed 2018].

[3] "Sqlite is Serverless," [Online]. Available: https://www.sqlite.org/serverless.html. [Accessed 2018].

[4] U. Okafor, R. K. Karne, A. L. Wijesinha and B. Rawal, "Transforming SQLITE to Run on a Bare PC," *Proceedings of the 7th International Conference on Software Paradigm Trends,* pp. 311-314, 2012.

[5] U. Okafor, R. Karne, A. Wijesinha and P. Appiah-Kubi, " A Methodology to Transform an OS-based Application to a Bare Machine Application," *The 12th IEEE International Conference on Ubiquitous Computing and Communications,* 2013.

[6] W. V. Thompson, H. Alabsi, R. K. Karne, S. Linag, A. Wijesinha, R. Almajed and H. Chang, "A Mass Storage System for Bare PC Applications Using USBs," *International Journal on Advances in Internet Technology,* vol. 9, pp. 63-74, 2016.

[7] W. Thompson, R. Karne, A. Wijesinha, H. Alabsi and a. H. Chang, "Implementing a USB File System for Bare PC Applications," *The Eleventh*

International Conference on Internet and Web Applications and Services, pp. 58-63, 2016.

[8] S. Soumya, R. Guerin and K. Hosanagar, "Functionality-rich vs. minimalist platforms: A two-sided market analysis," *ACM Computer Communication Review,* Vols. vol. 41, no. 5, pp. 36-43, 2011.

[9] L. He, R. K. Karne, A. Wijesinha and A. Emdadi, "Design and Performance of a Bare PC Web Server," *International Journal of Computers and Their Applications (IJCA),* 2008.

[10] G. H. Khaksari, A. L. Wijesinha, R. K. Karne, L. He and S. Girumala, "A peer-to-peer bare PC VoIP application," *4th IEEE Consumer Communications and Networking Conference (CCNC),* pp. 803-807, 2007.

[11] A. Alexander, A. Wijesinha and R. Karne, "A Study of Bare PC SIP Server Performance," *The Fifth International Conference on Systems and Networks Communications. ICSNC,* 2010.

[12] 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, 2009.

[13] F. G., Karne.R., W. A 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, 2009.

[14] 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,* 2010.

[15] 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.

[16] A. K. Tsetse, A. L. Wijesinha, R. K. Karne and A. Loukili, "A 6to4 Gateway with Co-located NAT," *IEEE International Conference on Electro Information Technology ,* 2012 .

[17] A. K. Tsetse, A. L. Wijesinha, R. K. Karne and A. Loukili, "A Bare PC NAT Box," *International Conference on Communications and Information Technology ,* 2012 .

[18] A. K. Tsetse, A. L. Wijesinha, R. K. Karne and A. Loukili, " Measuring the IPv4-IPV6 IVI Translation Overhead," *ACM Research in Applied Computation Symposium,* Vols. Measuring the IPv4-IPV6 IVI Translation Overhead , 2012.

[19] N. Kazemi, A. L. Wijesinha and R. Karne, "Design and Implementation of IPsec on a Bare PC," *2nd International Conference on Computer Science and its Applications (CSA),* 2009.

[20] B. S. Rawal, R. K. Karne and A. L. Wijesinha, "Split Protocol Client Server Architecture," *Seventeenth IEEE Symposium on Computers and Communications (ISCC),* 2012.

[21] B. Rawal, R. K. Karne and A. L. Wijesinha, "Splitting HTTP Requests on Two Servers," *The Third International Conference on Communication Systems and Networks COMSNETS,* 2011.

[22] B. Rawal, R. K. Karne and A. L. Wijesinha., " Mini Web Server Clusters for HTTP Request Splitting," *2011 IEEE International Conference on High Performance Computing and Communications,* 2011.

[23] P. Appiah-kubi, R. K. Karne and A. L. Wijesinha, "A Bare PC TLS Webmail Server," *International Conference on Computing, Networking and Communications, ICNC,* pp. 156-160, 2012.

[24] R. K. Karne, A. L. Wijesinha and S. Liang, "A Bare PC Mass Storage USB Driver," *International Journal of Comptuers and Their Applications,* 2013.

[25] A. Loukili, A. Tsetse, R. K. A. Wijesinha and P. Appiah-Kubi, " Performance of an IPv6 Web Server under Congestion," *12th International Conference on Networks (ICN),* 2013.

[26] A. Loukili, A. L. Wijesinha, R. K. Karne and A. K. Tsetse, " TCP's Retransmission Timer and the Minimum RTO," *21st International Conference on Computer Communications and Networks (ICCCN),* 2013.

[27] W. Agosto-Padilla, R. Karne and A. Wijesinha, "Insights into Transforming a Linux Wireless Device Driver to Run on a Bare Machine," *10th International Conference on Evaluation of Novel Software Approaches to Software Engineering,* 2015 .

[28] A. Alexander, A. L. Wijesinha and R. Karne, "Implementing a VOIP Server and a User Agent on a Bare PC," *The Second International Conference on Future Computational Technologies and Applications ,* pp. 21-26, 2010.

[29] S.Liang, R. K. Karne and A.L.Wijesinha, " A Lean USB File System For Bare Machine Applications," *the 21st International Conference on Software Engineering and Data Engineering, ISCA,* pp. 191-196, 2012.

[30] H. Chang, R. K. Karne and A. Wijesinha, "Migrating a Bare PC Webserver to Multi-core Architecture," *IEEE 40th Annual Computer Software and Applications Conference,* pp. 216-221, 2016.

[31] D. R. Engler and M. Kaashoek, "Exterminate all operating system abstractions," *Fifth Workshop on Hot Topics in Operating Systems,USENIX,* p. 78, 1995.

[32] "“The OS Kit Project," School of Computing, University of Utah, June 2002. [Online]. Available: http://www.cs.utah.edu/flux/oskit.. [Accessed 2018].

[33] V. S. Pai, P. Druschel and W. Zwaenepoel, "O-Lite: A unified I/O buffering and caching system," *ACM Transactions on Computer Systems,* vol. Vol.18 (1), pp. 37-66, 2000.

[34] J. Lange, Palacios and Kitten, " New high performance operating systems for scalable virtualized and native supercomputing," *24th IEEEInternational Parallel and Distributed Processing Symposium (IPDPS),* pp. 1-12, 2010.

[35] D. A. Patterson, G.Gibson and R. H. Katz, "A case for redundant arrays of inexpensive disks (raid)," *ACM,* p. 109–116, 1988.

[36] A. Brown and D. A. Patterson, "Towards availability benchmarks: A case study of software RAID systems," *the USENIX Annual Technical Conference (USENIX '00),* pp. 263-276, 2000.

[37] "6x USB Flash Drive Raid," April 2008 . [Online]. Available: http://analogbit.com/2008/04/21/6x-usb-flash-drive-raid/. [Accessed 2018].

[38] "100 Year Archive Task Force," January 2017. [Online]. Available: http://docshare01.doc hare.tips/files/12583/125832440.pdf.. [Accessed 2016].

[39] W. Thompson, R. K. Karne and A. Wijesinha, " Interoperable SQLite for a Bare PC," *13th International Conference Beyond Database Architectures and Structures,* pp. 177-188, 2017.

[40] G. H. Khaksari, R. K. Karne and A. L. Wijesinha, " A Bare Machine Application Development Methodology," *International Journal of Computers and Their Applications (IJCA),* Vols. Vol. 19, No.1, pp. 10-25, 2012.

CURRICULUM VITAE

NAME: Hamdan Ziyad Alabsi

PROGRAM OF STUDY: Information Technology

DEGREE AND DATE TO BE CONFERRED: Doctor of Science, 2018

EDUCATION

- Doctoral of Science in Information Technology
  Towson University, Towson, MD
  May 2018

- Master of Science in Information Technology Management
  St. Ambrose University, Davenport, IA May 2014

- Bachelor of Science in Business Management with a minor in Information Technology Management
  Montana State University, Bozeman, MT May 2012

Research interest

- Bare Machine Computing
- Database and distributed database
- Big data and E-commerce
- Cloud Computing

Publications

1. H. Z. Alabsi, W. V. Thompson, R. K. Karne, A. L. Wijesinha, R. Almajed, F. Almansour, *A Bare Machine RAID File System for USBs,* SEDE 2017: 26th International Conference on Software Engineering and Data Engineering, pp 113-118.

2. F. Almansour, R. K. Karne, A.L. Wijesinha, H. Alabsi and R. Almajed, *Middleware for NICs in Bare PC Applications,* 26th International Conference on Computer Communications and Networks (Poster Paper), ICCCN2017, Vancouver, Canada, 2017

3. W. V. Thompson, H. Alabsi, R. K. Karne, S. Linag, A.L. Wijesinha, R. Almajed, and H. Chang, *A Mass Storage System for Bare PC Applications Using USBs*,

International Journal on Advances in Internet Technology, vol 9, no 3 and 4, year 2016, pp 63-74.

4. W. Thompson, R. Karne, A. Wijesinha, H. Alabsi, and H. Chang, *Implementing a USB File System for Bare PC Applications*, ICIW 2016: The Eleventh International Conference on Internet and Web Applications and Services, pp 58-63.