2009 33rd Annual IEEE International Computer Software and Applications Conference

The Design and Implementation of a Bare PC Email Server

George H. Ford, Jr., Ramesh K. Karne, Alexander L. Wijesinha, and Patrick Appiah-Kubi
Department of Computer and Information Sciences
Towson University
Towson, Maryland, USA
gford3@students.towson.edu, rkarne@towson.edu, awijesinha@towson.edu, pappia1@students.towson.edu

***Abstract*— This paper presents the architecture, design and implementation of an email server that runs on a bare PC without an operating system or hard-disk. In addition to providing standard services offered by conventional email servers, the bare PC email server incorporates several unique features leveraging the absence of an operating system. For example, it implements novel algorithms for optimal multi-tasking, provides streamlined processing of messages enabling highly efficient integration of the SMTP and POP3 servers, and minimizes traditional software and protocol overhead. Additionally, it eliminates process overhead due to an operating system, offers enhanced security since the server is not vulnerable to attacks that target operating system flaws, and has a smaller code size. The complete server can be booted from removable media such as a USB drive. The bare PC email server demonstrates the ability of a self-contained, self-executing, complex software application to directly control the underlying PC hardware.**

Keywords- Bare Machine Computing (BMC), Application Object, Email Server, SMTP/POP3, Intertwining Protocols

I. Introduction

Conventional email servers are supported by an operating system (OS). They are unable to fully exploit the capabilities of the underlying hardware due to the OS, which acts as an intermediary. For example, an email server running on an OS-based system has increased overhead due to resource management, and complex concurrency control and task scheduling mechanisms. Moreover, it is difficult to minimize protocol overhead using an OS-based socket interface for communication with email clients and servers. Conventional email servers are also vulnerable to security holes in the OS. We describe a novel email server that executes on a bare machine without any OS. In Bare Machine Computing (BMC), an application object (AO) [8], executes directly on an Intel IA-32 based PC or device. An AO consists of the application program and all the necessary code to boot, load, and execute it, including direct interfaces to the hardware. Thus, the design and implementation of BMC applications in the form of self-contained and self-executable AOs is quite different from that of conventional OS-based or embedded system applications. Writing such applications presents a variety of software challenges, since there is no commercially available infrastructure for the development and deployment of such applications.

The focus of this paper is the architectural approach, design, and implementation of an SMTP-POP3 email server that executes on a bare machine. Like conventional email servers, the server can receive email, deliver email to valid local users, use SMTP relay forwarding to send email to the recipient's mail exchange domain for delivery by another email server, and interoperate with a variety of popular email clients and servers. In addition, the BMC email server architecture incorporates several novel features: 1) the ability to boot, load, and execute a complete email server application contained on a USB device; 2) management of persistent data storage on a USB device; 3) direct communication with a NIC; 4) a novel approach to optimal BMC-based multi-threaded task control; 5) support for multiple concurrent client sessions serviced on state-based, per-request transactions without interruption; 6) minimization of software and protocol overhead by streamlined processing of TCP and SMTP/POP3 messages and intertwined protocol implementation; 7) efficient integration of SMTP and POP3 server operations and SMTP relay forwarding within a single application; 8) enhanced security due to immunity against attacks targeting OS flaws; and, 9) smaller code size.

The remainder of this paper describes related work in section II, the architectural approach in section III, software design in section IV, software implementation in section V, and functional and operational testing in section VI.

II. Related Work

Many attempts have been made to eliminate OS abstractions, or provide a lean OS environment [1, 2, 11, and 13]. In [12], Linux is used to enable direct communication with the hardware. More recently, sandboxing techniques on the x86 systems [3] have extended OS kernels allowing applications to run guest plug-ins on the host OS.

Prior research related to email server architectures has focused on OS-based threading features. Threaded communications and processes can provide many advantages in terms of security and concurrent operations for users. However, many of the existing email server designs are lacking in some security-related features, such as spam and security awareness [10]. Other areas of prior research in email server architectures have included design focus on issues associated with scalability and dependability [5].

The BMC approach to developing applications is derived from dispersed operating system computing [6], and the AO concept. In the process of developing various AOs, techniques were devised to directly communicate with the hardware [7]. Other current BMC networking applications include a Web Server [4] and a VoIP softphone [9].

III. ARCHITECTURE

The BMC email server has been architected to include the necessary software structures to perform the server's intended functions. The server architecture is supported by common general-purpose hardware including USB-based bootable devices, network interface cards, USB-based persistent storage, and an Intel IA-32 based PC. The architecture uses BMC-based process threading techniques, circular lists, and stack mechanisms to provide efficient memory utilization and process control.

However, the architecture does not make use of any components of an OS. Likewise, the compilation and execution of the BMC application object for the email server does not make use of any system or API libraries of the compiler which would require an OS. The BMC approach only uses BIOS function calls to invoke the timer and read device addresses from the Intel Hub Controller. All other interrupts are managed by the Application Object as software interrupts.

A. Description

Fig. 1 shows the architecture of a BMC email server. For ease of reference, numeric labels are assigned to items within the figure. The server is initiated on a bare machine by a boot program (1) which is read from a USB-bootable device. The initial sector contains a bootstrap loader which loads the menu program (2), which in turn loads the AO. The AO starts by initializing various data structures, parameters, tasks, and objects. Control then passes to the Main Task (3). The basic BMC data structures used by the Main Task include a Circular Task List (12) and a Multiplexor mechanism (13) that switches between Received Tasks (18) and Circular List Tasks (SMTPS, POP3S, and SMTPC) and selects a running task (4). A TCP Table (TCB, 15) is used to store relevant information derived from the headers of TCP, IP and Ethernet messages (19). To communicate directly with the host Network Interface Card (NIC, 16), a BMC NIC driver was written (17). To perform the functionality of a POP3-SMTP email server, there are three components within the overall email server BMC Application. This includes an SMTP Server Object (5) to receive emails from clients, a POP3 Server Object (6) to deliver resident emails for local users' clients, and an SMTP Client (7) within the email server for SMTP Relay Forwarding when sending email to recipients in other email domains.

At initialization, a task pool of SMTP Server (8), POP3 Server (9), and SMTP Client (10) objects are created along with their associated TCB table entries for use by the server application. When one of these objects is in use to service a client-server connection, a reference to it is placed within the Circular Task List (12), and an active status flag is set within its associated TCB entry. The TCB entry contains all of the unique data attributes associated with the object, and its executable state information. When the task pooled object is inactive, the reference to the object is removed from the Circular Task list and pushed back onto its associated task pool's Task Stack (11). The active flag in the associated TCB entry along with associated data fields are then reset.

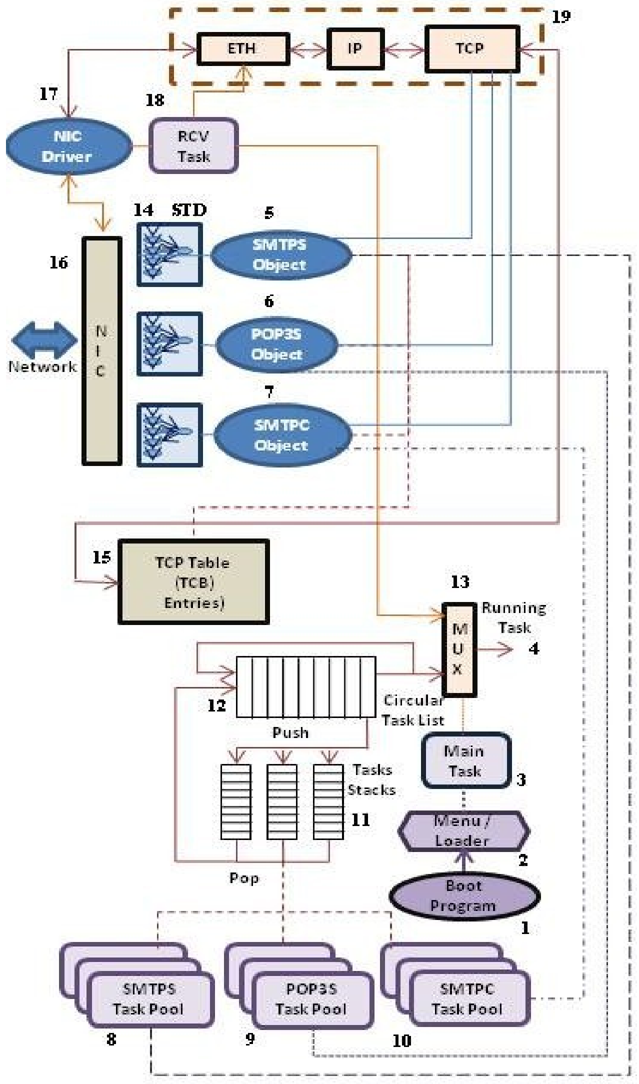


Figure 1. Email Server Architecture

B. *Novel Features*

The BMC email server provides multi-threaded task control that does not use message-based communications for inter-process communications as is done in many OS thread implementations. The server uses a simple optimal task scheduling algorithm that ensures the efficient processing of received and transmitted messages. The Main Task keeps a reference to active objects within the Circular List, which maintains information about their objects' data references within the TCB table entries for the object.

The BMC email server approach supports multiple concurrent client session connections. It achieves this by keeping track of the IP address and port number used for a client during session establishment. In case a client connection is lost or a connection remains idle for a specified time, the connection can timeout and the associated BMC email server objects are freed for some other subsequent client connection. An example of client-server interaction with intertwined TCP and SMTP protocols is shown in Fig. 2.

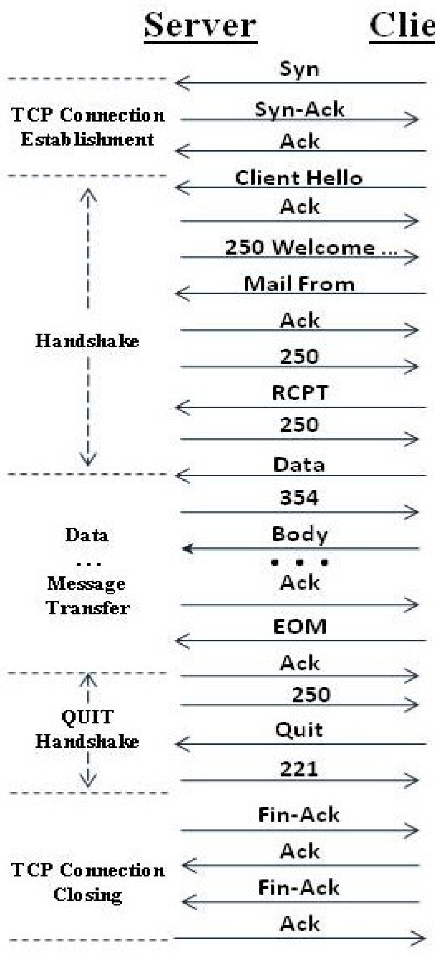


Figure 2. Intertwined Network and Application Protocols

Intertwining network and email protocols in a BMC email server expedites the handling of Ethernet, IP and TCP headers and the processing of SMTP and POP3 messages within a single thread of execution without interruption. Understanding how to intertwine protocols in this manner involved analysis of operational email systems using a Wireshark packet analyzer. Intertwining reduces the overhead due to typical protocol layering and context switching. Thus, a transaction request with data is an atomic transaction serviced within a single active thread of execution from receipt of a request until generation of a response. This approach also has the benefit of minimizing the software overhead associated with process handling in layered protocol approaches. In the BMC email server, interactions due to TCP / SMTP protocol intertwining (Fig. 2) are all coded within a state-diagram-driven approach (Fig. 1, item 14), where each state corresponds to a particular SMTP command, triggering the necessary SMTP and TCP actions.

IV. Design

The design of the BMC email server avoids the use of any OS-based features, and creates an application which will run on a bare machine. The user interface for the BMC email server is text-based, which provides some configuration parameters to be specified at startup. The design goal of the BMC email server was its interoperability with BMC email clients, conventional email clients such as MS-Outlook, Eudora, Mozilla, Thunderbird, Pegasus, as well as other email servers (for SMTP relay forwarding).

In order to perform a proof-of-concept on the initial design of the email server and to ensure correct software functional design steps of the email operations, a Java-based email server was designed first. This had a secondary benefit in addition to conceptual verification of protocols and interaction of clients and server message exchanges. When initially testing, the BMC email server was isolated on a research lab LAN environment to insure a controlled testing environment. The first email server used as the target for SMTP message forwarding was the Java-based email server prototype on the research lab LAN. The Java-based prototype provided the first "remote domain" with which the BMC email server communicated for SMTP Relay Forward testing. Once proven to operate as designed, the BMC email server was connected to the campus Intranet and finally to the Internet for testing.

During the design of the BMC email server, each SMTP and POP3 command was prototyped using State Transition Diagram Modeling (Fig. 1, item 14). Each state was modeled to indicate SMTP and TCP message protocol transitions (Fig. 3) during client and server protocol handshaking. A typical implementation of the HELO state in C++ is shown in Fig. 4. Notice that the Send-ACK and Send 250 illustrate the BMC email server's ability to intertwine protocols.

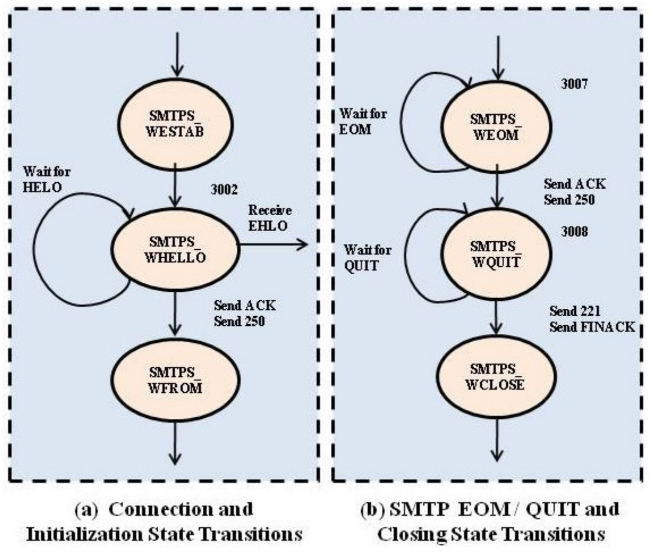
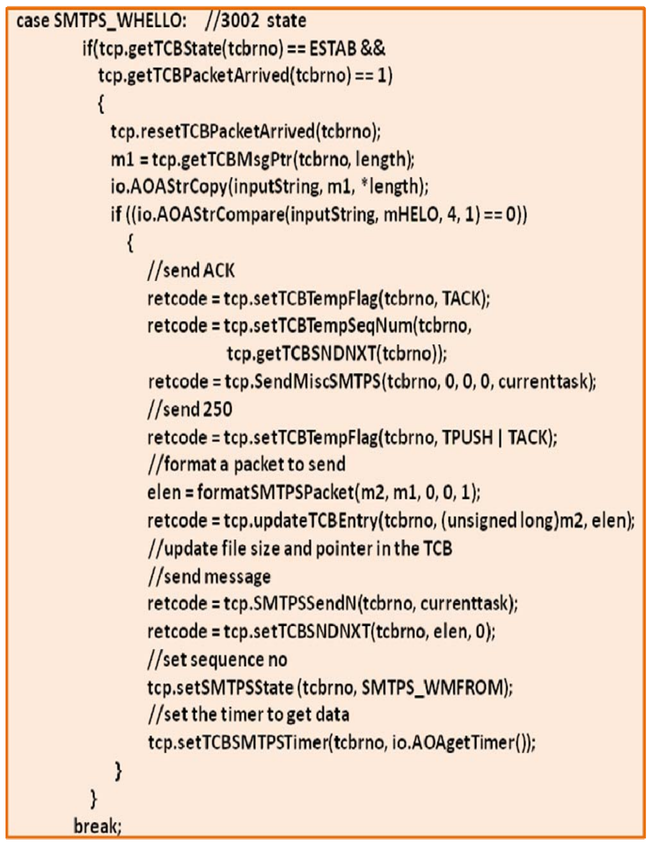


Figure 3. Intertwined Protocol State Transition Examples



```
case SMTPS_WHELLO:   //3002 state
          if(tcp.getTCBState(tcbrno) == ESTAB &&
            tcp.getTCBPacketArrived(tcbrno) == 1)
            {
              tcp.resetTCBPacketArrived(tcbrno);
              m1 = tcp.getTCBMsgPtr(tcbrno, length);
              io.AOAStrCopy(inputString, m1, *length);
              if ((io.AOAStrCompare(inputString, mHELO, 4, 1) == 0))
                {
                  //send ACK
                  retcode = tcp.setTCBTempFlag(tcbrno, TACK);
                  retcode = tcp.setTCBTempSeqNum(tcbrno,
                              tcp.getTCBSNDNXT(tcbrno));
                  retcode = tcp.SendMiscSMTPS(tcbrno, 0, 0, 0, currenttask);
                  //send 250
                  retcode = tcp.setTCBTempFlag(tcbrno, TPUSH | TACK);
                  //format a packet to send
                  elen = formatSMTPSPacket(m2, m1, 0, 0, 1);
                  retcode = tcp.updateTCBEntry(tcbrno, (unsigned long)m2, elen);
                  //update file size and pointer in the TCB
                  //send message
                  retcode = tcp.SMTPSSendN(tcbrno, currenttask);
                  retcode = tcp.setTCBSNDNXT(tcbrno, elen, 0);
                  //set sequence no
                  tcp.setSMTPSState (tcbrno, SMTPS_WMFROM);
                  //set the timer to get data
                  tcp.setTCBSMTPSTimer(tcbrno, io.AOAgetTimer());
                }
            }
         break;
```

Figure 4. Example Source Code for Client-Server HELO Handshake

The client sends HELO and the server responds by sending the TCP ACK and also the SMTP 250 response indicating its identity. Similarly, the SMTP EOM, and QUIT commands are intertwined with TCP operation. In this case, when the client sends the message EOM, the server responds with both the TCP ACK and the SMTP 250 OK response. Also, when the client sends a subsequent QUIT command, the server responds with both a SMTP 221 server closing response and a TCP FINACK handshake. The FINACK closes the TCP/IP connection between the client and server. These state diagrams exemplify the intertwining of SMTP and TCP protocols within the session, demonstrating the efficiency achieved by the design and implementation of a BMC Email Server. In contrast, the SMTP and TCP protocols in a conventional email server communicate through a standard socket interface and the protocol modules are independent, which results in a less efficient implementation.

V. IMPLEMENTATION

The implementation uses a standard MS-Windows environment, Visual C++, and the MASM Assembler for software development. However, this approach does not include any OS-related libraries and system calls. Instead, the Application Object (AO) uses direct hardware interfaces design for Bare PC computing [7]. Most of the direct hardware interfaces are implemented in assembly language using software interrupts. The size of this assembly code is approximately 1,800 lines. These direct hardware interfaces include display, keyboard, timers, task management, and real/protected mode switching. The 3COM 905CX NIC driver code consists of approximately 1400 lines of assembly code with the rest of the code written in C. Similarly, the USB driver uses approximately 133 lines of assembly code with the rest of the code written in C. The implementation of the email server architecture depicted in Fig. 1 is written in C++ in an object-oriented fashion. The BMC server contains 31,743 lines of code including 10,000 lines of comments, and 14,000 executable statements. The resulting single monolithic executable (AO) is 512 sectors of code (256 KB).

The software is placed on the USB and includes: the boot program, startup menu, AO executable, and the persistent file system (used for User Profiles, emails, and attachments). The USB containing this information is generated by a tool (designed and run on MS-Windows) that creates the bootable BMC application for deployment. The tool, which generates the boot load sector, and copies the executable and associated files to the USB, consists of only 469 lines of C++ code.

A. *Bare Machine Computing Internals*

Due to the absence of an OS in BMC applications, several implementation challenges must be addressed when designing an application. The application itself must manage tasking, scheduling, memory map use, interrupts, I/O handling, and concurrency control. For example, the AO programmer controls task creation, execution, and termination. Pools of tasks are created for each required entity (e.g. SMTPS, POP3S, and SMTPC) during initialization. In this system, up to 4000 tasks have been created. When a client request arrives, a given task is pulled from the task stack and inserted into the circular list to run. A completed task is returned to its stack. Notice that the same task will be reused by a new request, without the need to recreate the task. This requires a never ending while loop in each task (when a task is complete, it always returns to the top of the while loop). Inside the while loop, a process request is called which handles the complete execution of the request. The state transition flow of the process request call will only return to the task when the process is complete. This is a very efficient way to implement tasks without incurring creation and termination overhead. When the AO application finishes its job, all the current running tasks are terminated and returned to the main task, which in turn returns to the user menu.

The BMC approach requires that the AO programmer makes scheduling decisions at coding time. Thus, when a program requires resources, the programmer suspends with a delay, and when a corresponding signal arrives for this task or when the delay timer expires, the task is resumed. The suspend mechanism uses state and control information stored in the TCB. The TCB mechanism allows reentrant code, since the task instance state, and control variables are stored in the TCB entry. The tasks that are waiting to be executed are placed in the circular list and processed in FIFO (first in first out) order. The main task always runs and controls the execution of Receive Task and other tasks (SMTPS, POP3S, and SMTPC). The Receive Task has higher priority than the other tasks. This task mechanism has been shown to result in optimal performance for the case of a BMC Web server [4] and it is expected to be optimal for other servers, including the BMC email server.

A novel approach is used for task execution in BMC applications. When a task starts executing, it will use a single thread of execution that involves many protocol layers and processes. It will return to the caller after completion, or suspend itself if it is unable to continue execution. The CPU is continually kept busy during task execution. This self-controlled, self-managed technique for BMC task execution and scheduling has proven to be very efficient for Web server implementations [4].

B. *Storage, Interrupts, User Interfaces and Networking*

The AO programmer also manages the memory required for a given application. BMC memory is all real-mode, with persistent storage in a USB mass storage device or on the network. The AO programmer lays out a memory map prior to implementing a given application and manages both static and dynamic memory. The code, data, stack, and dynamic memory areas are pre-allocated with validation occurring at run-time.

BMC interrupt handling for a process is reduced to an absolute minimum. Hardware timer, NIC, and keyboard interrupts are the only hardware interrupts allowed during a single thread of execution. All other interrupts are implemented in software and controlled by the AO programmer.

BMC I/O user interfaces (such as display, network, and USB) are currently text-only since graphical user interfaces were not essential for the networking and server applications developed so far. The BMC device driver interfaces for standard 3COM NICs, and USB drivers for keyboard, mouse, and mass storage are different from standard OS-based drivers. The USB interfaces can work with a USB hub, thus making BMC systems independent of motherboard connector configuration and internal I/O preferences.

C. *Development Environment Challenges*

Numerous other challenges are faced with respect to the development environment, debugging, and testing of BMC applications. Some tools have been developed to debug and test BMC application code. These tools assist in monitoring the use of memory, memory inspection, placing debug messages into screen memory for display, and monitoring of network traffic and protocols. These tools also provide the ability to use batch driven files for compilation and linking, boot and load programs for USB, memory dump utilities for memory display, trace tools, and exception and trap mechanisms to take control during AO execution.

Tools freely available on the Internet including some from SourceForge as well as others developed in-house were used in the design, verification, integration, and testing of the BMC Email Server implementation. Commercial email clients were tested for compatibility. Also, functional and operational comparisons were made to other email servers to validate proper email server actions and responses.

VI. Functional and Operational Testing

After creating the email server AO in the laboratory, it was functionally tested using a variety of email clients. The client-server interactions were captured and studied using Wireshark. These sessions were then analyzed and validated to ensure correct sequencing of SMTP and POP3 with TCP intertwining.

The operational test scenario included the creation of hundreds of email users for the system, with enough initial storage for several emails per user. Each email is capable of carrying multiple MIME-encoded attachments. This testing included several interactive (human) and automated clients concurrently sending and retrieving email via SMTP and POP3, respectively, after authentication. BMC, Java, MS-Outlook, Mozilla Thunderbird, Eudora, Pegasus, SendMail, and Pine email clients were included in testing. Client machines included a mix of BMC, MS-Windows, and Linux based systems. The tests also included programmed "batch sending" of large numbers of email sends/retrievals in quick succession by both BMC and Java clients. These tests led to further enhancements of the email server to accommodate variations noted in Wireshark captures of client-server interactions related to protocol handshaking on session closure and the inclusion of optional application commands used by some commercial email clients. This resulted in a more robust email server that can interact with a wide variety of clients, and conduct SMTP relay forwarding with several commercial mail exchange servers.

The functional and operational tests have shown that the BMC email server, with its intertwining of protocols, successfully implements the expected capabilities of an email server. It has robust handling of concurrent command-response interactions for simultaneous email client sessions, is capable of SMTP relay forwarding, and provides error handling for sustained operations. Additional testing has revealed further benefits. Since the BMC email server executes on a machine which does not have any OS, and hence has no OS-based components, it is not vulnerable to certain types of Web-based security risks and attacks.

VII. Conclusion

This research has demonstrated that complex client-server based application objects can be developed using the BMC environment without any host OS for execution. Similar to conventional servers running on OS-based systems, the BMC email server can support SMTP / POP3 clients and SMTP Relay Forwarding. The Bare PC email server design also includes the following significant and novel features: USB-based persistent storage of emails, user validation and authentication, TCP and SMTP / POP3 intertwining modeled as state driven action-request responses, a single thread of execution, and timed request-response protocol interactions.

Extensive testing shows that the server can interoperate with a variety of email clients and servers. In addition, the BMC email server is immune to attacks that target vulnerabilities in specific OS platforms. The experience gained with building the BMC email server has also led us to explore BMC concepts and designs for VoIP, secure teleconferencing (voice and whiteboard), and webcam video-conferencing as part of on-going BMC research conducted at Towson University.

Acknowledgment

We are grateful to the late Dr. Frank Anger whose support of the AO (Application Object) Architecture project during his tenure as a director at the National Science Foundation resulted in the awarding of an exploratory research grant. This early work was the forerunner of dispersed operating system computing and BMC research.

References

[1] D. R. Engler and M.F. Kaashoek, "Exterminate All Operating System Abstractions," 5th Workshop on Hot Topics in Operating Systems, USENIX, Orcas Island, WA, May 1995, p. 78.

[2] B. Ford, M. Hibler, J. Lepreau, R. McGrath, and P. Tullman, "Interface and Execution Models in the Fluke Kernel," Proceedings of the 3rd Symposium on Operating Systems Design and Implementation, USENIX Technical Program, New Orleans, LA, February 1999, pp. 101-115.

[3] B. Ford, and R. Cox, Vx32: "Lightweight User-level Sandboxing on the x86", USENIX Annual Technical Conference, USENIX, Boston, MA, June 2008.

[4] L. He, R. K. Karne, and A. L. Wijesinha, "Design and performance of a bare PC web server," International Journal of Computer and Applications., Acta Press, June 2008.

[5] E. Kageyama, C. Maziero, and A. Santin, "A Pull-based Email Architecture," Proceedings of the 2008 ACM symposium on Applied Computing, ACM, Fortaleza, Ceara, Brazil, 2008, pp. 468-472.

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

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

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

[9] G. H. Khaksari, A. L. Wijesinha, R. K. Karne, L. He, and S. Girumala, "A Peer-to-Peer Bare PC VoIP Application," Proceedings of the IEEE Consumer and Communications and Networking Conference, IEEE Press, Las Vegas, Nevada, January 2007.

[10] A. Pathak, S. Roy, and Y. Hu, "A Case for a Spam-Aware Mail Server Architecture," 4th Conference on Email and Anti-Spam (CEAS) 2007, Microsoft Corporation, Mountain View, CA, August 2007.

[11] "Tiny OS," Tiny OS Open Technology Alliance, University of California, Berkeley, CA, 2004, http://www.tinyos.net/

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

[13] "The OS kit project," School of Computing, University of Utah, Salt Lake, UT, June 2002, http://www.cs.utah.edu/flux/oskit