A Simple UDP-Based Web Server on a Bare PC with 64-bit Multicore Processors: Design and Implementation

Navid Ordouie
Computer and Information Sciences
Towson University
Towson, MD
nordouie@towson.edu

Ramesh K. Karne
Computer and Information Sciences
Towson University
Towson, MD
rkarne@towson.edu

Alexander L. Wijesinha
Computer and Information Sciences
Towson University
Towson, MD
awijesinha@towson.edu

Nirmala Soundararajan
Chemistry, Computer and Physical Sciences
Southeastern Oklahoma State University
Durant, OK
nsoundararajan@se.edu

***Abstract*—Most Web transfers use TCP, notable exceptions being those that use HTTP/3 over QUIC and CoAP in the IoT, which require UDP. The latter protocols are complex because they provide essential functionality for real-world Web applications. We describe the design and implementation of a UDP-based simple Web server that runs on a bare PC with 64-bit multicore processors. Previously, we built UDP and TCP-based bare PC Web servers that run on 32-bit and 64-bit multicore processors respectively. The present design improves, modifies, and integrates the previous designs. We compare delays for designs with and without a last ack by capturing a single HTTP request and reply between the Web server and a client connected through an Ethernet hub. This work is a first step towards building robust and scalable bare PC Web servers that can transfer data efficiently and securely to browser applications in the real-world.**

Keywords—Bare PC, Bare Machine Computing, 64-bit processor, multicore processor, Web server, UDP

I. INTRODUCTION

Client/server computing on the Web commonly uses TCP. The overhead due to TCP can be eliminated by using the QUIC protocol over UDP [RFC 9000]. QUIC is a transport protocol that deals with reliability, error correction, and security. The specification of HTTP/3 based on QUIC [RFC 9114] reflects current interest in using UDP for reliable secure Web data transfers. QUIC can also be used for unreliable data transfers [RFC 9221]. CoAP used for M2M applications in the IoT [RFC 7252] also typically uses UDP for transport, although CoAP can also run over TCP [RFC 8323]. Both QUIC and CoAP require some form of operating system or kernel support. We describe a simple UDP-based protocol for HTTP transfers between bare machines with no operating system or kernel, where the server runs on 64-bit multicore processors. Bare Machine Computing (BMC) systems run general-purpose computing applications and are not the same as embedded systems or bare metal Linux systems. Due to their reduced attack surface, bare machines are especially suited for secure real-world environments. Bare machines and BMC applications are discussed later in the paper.

This UDP-based protocol for Web transfers has no last ACK. Earlier versions of a UDP-based protocol with a last ACK are described for a 32-bit multicore BMC Web server in [1] and for a 32-bit single-core BMC Web server in [2]. When there is no last ACK, all data packets are not sent at one time by the server. Instead, the data file is split, and a limited amount of data in a small number of packets is sent at a time to the client. After receiving these packets, the client makes a new request to receive the next set of packets. In addition to making the server stateless and simpler, possible benefits of this protocol include elimination of server retransmissions due to timeouts and improved security due to limiting the amount of data sent at a time. The protocol may also be useful for resource constrained servers. Potential protocol benefits are traded off versus increased client complexity. We do not discuss tradeoffs in this paper.

Using Wireshark packet captures, we found that the server completes a typical HTTP transfer of a 4 KB file about 9 times faster without a last ACK because there is no waiting. We also briefly review the multicore server architecture and examine shared single NIC and Web server load balancing issues, while noting a few 32 to 64-bit translation issues relevant to the present server design.

The main contribution of this work is a simple UDP-based protocol for efficient HTTP transfers between stateless 64-bit multicore bare servers and clients. The rest of this paper is organized as follows. Section II introduces the new UDP-based protocol for Web transfers, Section III provides an overview of BMC. Section IV briefly discusses related work. Section V describes the 64-bit multicore server architecture and design. Section VI gives preliminary results for delays with and without the last ACK. Section VII concludes the paper.

II. A Simple Protocol for UDP-Based Web Transfers

The UDP-based protocol for HTTP transfers between bare machines has no relation to QUIC or CoAP. It can be configured to work with or without a last ACK. As Fig. 1 shows, the UDP-based protocol eliminates the TCP overhead due to establishing and closing connections, unnecessary retransmissions, and TCP ACKs. If more packets are needed or packet loss occurs, the client decides when to send another GET request. An HTTP/PHP request to a bare server is associated with a file name and attributes combined and referred to as the GET request. POST requests are similar. The GetACK response from the server contains the file size and number of data packets to be sent giving the client complete details about the file. In Fig. 1(b)), the server sends all the packets and waits for the last ACK. In Fig. 1(c)), the server only sends *n* (e.g., *n*=4) packets at a time, avoiding the last ACK and all retransmission related overhead and logic. When there are more packets to be sent, the client sends a GET request with the next packet number. For large files, multiple transfers of *n* packets may be needed. If a packet is lost, a client will request all *n* packets again. This approach simplifies the server but adds some more logic at the client. Packet numbers are used to handle out-of-order packets. The parameter *n* can vary based on file size and other design criteria.

The only overhead in this approach is due to additional GETs and GetACKs. This overhead is offset by reduced server complexity. The server is effectively stateless for a given client request. After initialization, the client sends only the GET request and the server sends only the data. After sending a set of packets, the server does not interact with the client until the client requests more packets. There is no waiting at either end other than waiting for the GetACK at the client. Although we did not use DTLS [RFC 9147] to secure communications between the client and the server, this protection can be easily added in a future version of the protocol (security protocols such as TLS, IPsec and SRTP have been implemented with other bare machine applications).

III. Bare Machine Computing

The Application-Oriented Architecture (AOA) [3] and its successor Dispersed Operating System Computing (DOSC) [4] evolved into Bare Machine Computing (BMC) [5]. Prior publications on BMC (or bare PC/bare) applications are archived at the website in [6]. The essential differences between conventional computing based on an Operating System (OS) and computing based on the BMC approach are shown in Fig, 2. In the BMC approach, a computing device is made bare: it has no OS and no hard disk, and only uses the BIOS in the boot process. Bare applications can run on older or newer x86 and x64 compatible Intel processors. The application software is written in C/C++ with a small amount of assembly code. A BMC system is based on a single programming environment. The boot, loader, and interrupt code are written in assembly. One or more applications can be compiled as an application suite to generate a single monolithic executable. This is statically compiled and linked with no external software or libraries.

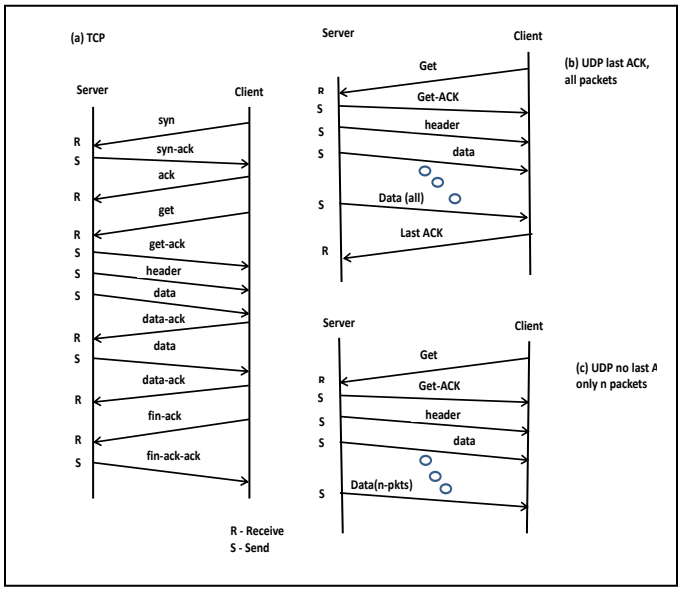

Fig. 1. Message exchanges for HTTP transfers over TCP and UDP.

The BMC paradigm was motivated by its potential to eliminate the disadvantages of conventional OS-based computing. BMC executables are small and include only the necessary bare drivers and minimal code to support other functions used to run a given application suite. There are no kernel or system calls, and no OS-related management functions. The application programmer has full control of the hardware and required interfaces, and the application suite directly communicates to the hardware without any middleware via a hardware API (HAPI). The bare application software is user controlled and physically secured by placing it on external storage. Currently, this is a USB flash drive owned by the user, who is responsible for guaranteeing its security. Bare applications only run intended functions with no options to introduce new functionality. For example, attackers

cannot run scripts, create additional threads, or open new ports. Bare machine applications have no OS-related vulnerabilities and eliminate overhead due to the OS.

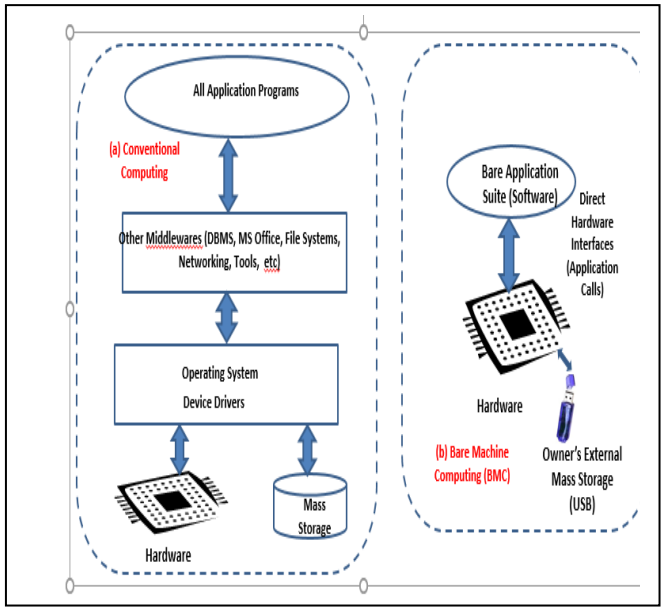

Fig. 2. Conventional versus Bare Machine Computing (BMC).

Moreover, conventional OS functions are not duplicated in an application suite as there is no centralized kernel running in the system. A typical OS provides services for all applications, while a bare machine application suite is designed to run only a desired small set of applications. The bare-to-bare communication is implemented as application-to-application avoiding all middle layers. There are no heterogeneous components and no layers in the application suite. An application suite image is typically very small since BMC systems are designed for domain-specific and intended functionality (e.g., a TCP Web client code image is less than 200 KB and a TCP server with four applications is less than 1 MB). The assembly code is minimal in BMC applications (e.g., a USB 2.0 driver has 24 lines of assembly code including comments and only two functions).

IV. Related Work

The design of 32-bit multicore UDP and TCP-based bare Web servers are described respectively in [1] and [7]. An attempt to migrate a 32-bit single core bare Web server to a 64-bit multicore architecture was made in [8]. That work was preliminary, focused primarily on migration, and did not use UDP. Exokernel [9], Microkernel [10], Tiny-OS [11], IO_Lite [12], OS-Kit [13], and RIOT [14] are a few examples of numerous approaches to reduce the size and complexity of the OS or kernel, give more direct hardware access to applications, or move some OS functions into user space. These approaches differ from BMC in that some form of an OS or kernel is needed. To the best of our knowledge, we are unaware of other work that enables general-purpose computing applications to run without an OS or kernel.

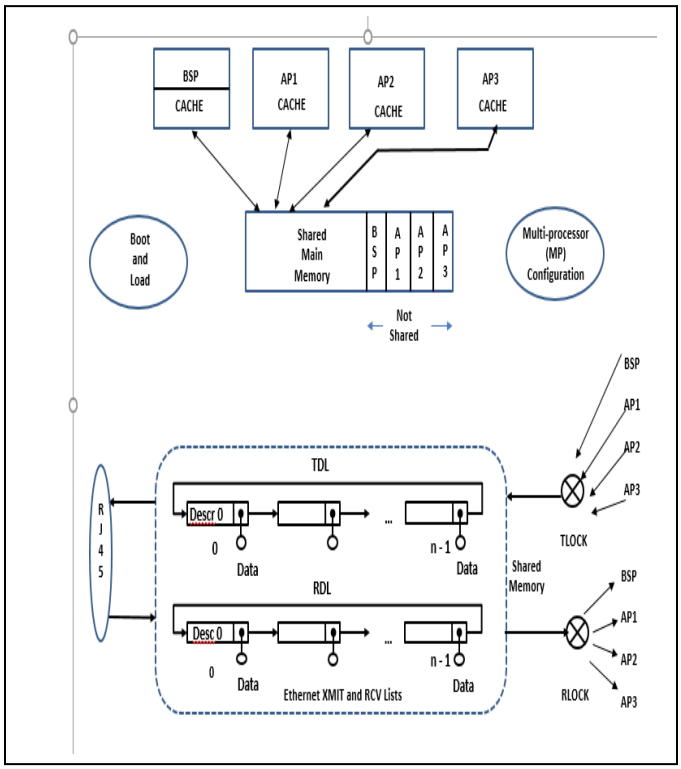

Fig. 3. Bare multicore server architecture.

V. 64-Bit Multicore Web Server

The bare Web server runs on a Dell Optiplex 9010 desktop with four 64-bit cores, 4 GM main memory and a 1 gigabit Intel Ethernet (1GbE) network adapter. There is no OS or kernel running in the machine. Fig. 3 shows the bare server architecture. The four cores are named BSP (boot processor), and AP1, AP2, and AP3 (application processors). The BMC multicore server design for 32-bit processors is described in [1]. The 64-bit multicore server was derived from this 32-bit multicore server. For BMC applications, most difficulties in translating 32-bit code to 64-bit code were related to 64-bit data and 64-bit addresses. We also modified BMC device drivers to work with 64-bit data values. This modification required checking the relevant C++, C and ASM code. The 64-bit multicore design differs significantly from the 32-bit design. Architectural issues in multicore are usually dealt with at the OS level. In BMC application development, such issues must be resolved during design and implementation. They include shared memory, locking, cache, shared single Network Interface Controller (NIC), Web server load balancing, and multicore configuration. Only shared single NIC, Web server load balancing issues, and design details relevant to processing requests are discussed here.

A. Shared Single NIC

The bare NIC driver is integrated with bare application. It has two paths, one each for transmit and receive. It also has two data structures (circular lists), known as the transmit descriptor list (TDL) and receive descriptor list (RDL). These lists are stored in main memory and used by the bare application and the driver to communicate. There can be 4096/8192 descriptors defined for these circular lists. When a packet arrives, the DD bit in the RDL descriptor is set. When a packet is received, the driver resets the DD bit. Similarly, when a packet is transmitted, the hardware sets the DD bit in the transmit descriptor. The driver can reset the transmit descriptor DD bit and move on to the next slot. The management of circular lists is done in the driver through IN and OUT pointers for TDL and RDL. SendINPtr, SendOUTPtr, RcvOUTPtr, and RcvINPtr need locking to modify them. When a packet arrives, RcvINPtr is used to store the packet in the list. It is incremented after inserting the packet. RcvOutPtr is used to read the packet inserted; it is incremented after the packet is discarded. Similarly, SendINPtr is used to form a packet; it is incremented for the next packet. SendOUTPtr is incremented after the packet is transmitted. The bare application manages these pointers using a locking HAPI with C++, C and ASM paths to lock and unlock shared memory.

B. Web Server Load Balancing

A client request is processed by the Web server in a single thread of execution with or without a last ACK. In Fig. 1(b), the server must keep state for each request until it gets the last ACK, which ends that request. In Fig. 1(c), the server does not keep state. It sends Get-ACK and data, but there is no waiting because there is no last ACK. As there is only one RDL, the core that gets a GET request processes the request. This poses a challenge for load balancing in a multicore architecture. In Fig. 1(b), the last ACK may be received by any core, and it may need to redirect that ACK to the appropriate core. Alternatively, one core could transmit and receive and delegate to other cores as implemented in [7]. This requires buffering and locks at each buffer.

With a last ACK, many requests are waiting in the queue to receive the last ACK for completion. With no last ACK, after sending GetACK, *n* packets can be sent in a single thread of execution. There is no need for queueing client requests. As the client/server interactions are few, there is not much to parallelize with respect to client requests in BMC applications. The only parallelism that can be exploited is dispatching client requests to all cores evenly. However, as there is only one NIC (single receive and transmit), this will become a bottleneck in achieving higher performance. In the bare Web server, the starting point for a client request is a GET message. As there is only one receive flow control in the NIC and there are four cores that can access an incoming GET packet, load balancing becomes critical. One approach is to lock the RDL descriptor and let one core receive the GET packet.

The bare server uses an alternative approach that avoid locks and their complexity. As there are only four cores in the server, a software approach based on round robin with flags is used to avoid concurrency control. Each core has its own ready flag: io.bspready, io.ap1ready, io.ap2ready, and io.ap3ready. There is one common token, io.token, indicating which core currently has a token. The token can be 0x100, 0x101, 0x102, 0x103 for BSP, AP1, AP2, AP3 respectively at any given point in time. When a token is at hand for a core, that core grabs the token, and gives the token back to the next core in round robin fashion, if that core is ready. There is only one io.token variable in memory, so only one core at a time can get to memory to modify the token, which guarantees mutual exclusion. When a core receives a token, it turns off its ready state, processes the GET request, and then sets its ready state to 1. When a core receives a GET packet, it processes the request. In this approach, each core has a chance to get the request, thus redistributing the load to all cores. At the transmit end, all cores use locks to send data, thus creating a bottleneck in sending data. Two or more NICs could be used to improve performance as in [15].

C. Processing Requests

When processing a request with a last ACK (Fig. 1(b)), the control flow for 64-bit is the same as the control flow for 32-bit in [2]. When a packet arrives, it triggers RcvCall() as a single thread of execution in an event-based system. This is done in all BMC applications to eliminate the need for an OS or kernel. As in [1], we added a 16-byte data header that has control information including a 4-byte request id, 4-byte core id, size of pkt/total size, total packets/pktno. The last two fields are interpreted differently depending on the packet type. This header is sent with the GetACK enabling the client to know all the resource file details. Some parts of the header simplify the multicore architecture and packet management at the server. Reqid is used at the server as an index to a linear list that contains all the details about the client and the resource file. This id is a monotonically increasing number that wraps around. Indexing avoids searching to find a client request in the list. The coreid helps the server to manage requests by indicating which core is processing the request. When the last ACK arrives with the 16-byte data header, the server plugs in the same reqid and coreid in this header. The client also plugs in total packets received in the pktno and the total bytes received in the size field. This structure reduces the complexity to manage packets between the server and the client. The linear list for holding client requests is managed and maintained until client sends the last ACK. Each client's request state is also managed in this list.

When processing a request in the protocol without the last ACK (Fig. 1(c)), the control flow is the same as above except for the following. To process the UDP request in processUDPRequest(), only the GET control flow activates since there is no last ACK. In this case, there is no linear list to maintain client requests and no state maintained for each request. The 16-byte data header is still used to simplify request management and multicore setup for load balancing. The data header is used by clients to request subsequent GETs for continued data if the server only sends partial data for a large file. Eliminating the last ACK reduces complexity and makes it easier to design, implement and debug the protocol.

VI. Preliminary Measurements

To find HTTP request completion times at the server with and without a last ACK, we used packet data captured by Wireshark running on a Windows machine connected to the Ethernet hub. For 4 KB files, typical times were as follows (these were found by using the packets shown in Fig. 4). With the last ACK (port 29539), request completion time = 3.241721 – 3.238404 secs = 0.003357 secs; and with no last ACK (port 7020), request completion time = 3.1094 - 3.1091 secs = 0.0003 secs. While the server completes the request without a last ACK about 9 times faster, the time for the client to receive and process all the packets is not known in this case. More accurate and detailed performance measurements could be obtained in future by also measuring the request delay at the bare machine client; and using multiple requests and multiple clients to fully utilize the bare machine server.

VII. Conclusion

We described the design and implementation of a simple UDP-based 64-bit multicore Web server that runs on a bare PC. Two versions of the UDP-based protocol for HTTP transfers with and without a last ACK were compared. Preliminary measurements confirm that the server can complete a request much faster when it does not have to wait for a last ACK. In this case, the server is stateless and only interacts with a client when the client makes another request for more data. This in turn may improve overall server performance. We also discussed shared single NIC and Web server load balancing issues. More research is needed to investigate the protocol with respect to security, scalability, and server performance under heavy loads.

References

[1] N. Ordoui, N. Sounderarajan, R. K. Karne, and A. L. Wijesinha, "Developing Computer Applications without any OS or Kernel in a Multi-core Architecture," International Symposium on Networks, Computers and Communications (ISNCC), 2021.

[2] N. Soundararajan, R. K. Karne, A. L. Wijesinha, N. Ordouie, and B. S. Rawal, "A Novel Client/Server Protocol for Web-based Communication over UDP on a Bare Machine," 18th Student Conference on Research and Development (SCOReD), 2020.

[3] R. K. Karne, "Object-oriented Computer Architectures for New Generation of Applications," Computer Architecture News, Vol. 23, No. 5, December 1995.

[4] R. K. Karne, K.V. Jaganathan, T. Ahmed, and N. Rosa, "DOSC: Dispersed Operating System Computing," 20th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), Onward Track, 2005.

[5] U. Okafor, R. Karne, A. Wijesinha, and P. Appiah-Kubi, "Eliminating the Operating System via the Bare Machine Computing Paradigm," 5th International Conference on Future Computational Technologies and Applications (Future Computing), 2013.

[6] Bare Machine Computing, http://orion.towson.edu/~karne/dosc/pubs.htm. Accessed Oct. 15, 2022.

[7] N. Soundararajan, R. Karne, A. Wijesinha, N. Ordouie, and H. Chang, "Design Issues in Running a Web Server on Bare PC Multi-Core Architecture," 44th Annual Computers, Software, and Applications Conference (COMPSAC), 2020.

[8] H. Chang, R. K. Karne, and A. Wijesinha, "Migrating a Bare PC Web Server to a Multi-core Architecture," 40th Annual International Computer Software and Applications Conference (COMPSAC), 2016.

[9] D. R. Engler, "The Exokernel Operating System Architecuture," Ph.D. thesis, MIT, 1998.

[10] I. Odun-Ayo et al, "An Overview of Microkernel Based Operating Systems," IOP Conference Series: Materials Science and Engineering, 1107 012052, 2021.

[11] TinyOS Home Page, http://www.tinyos.net/, Accessed Oct. 15, 2022.

[12] V. S. Pai, P. Druschel and W. Zwaenepoel, "IO-Lite: "A Unified I/O Buffering and Caching System," ACM Transactions on Computer Systems, Volume 18, Issue 1, February 2000.

[13] The OSKit Project, http://www.cs.utah.edu/flux/oskit, Accessed Oct. 15, 2022.

[14] RIOT – The friendly Operating System for the Internet of Things, https://www.riot-os.org, Accessed Oct. 15, 2022.

[15] F. Almansour, R. K. Karne, A.L. Wijesinha, and B. Rawal, "Ethernet Bonding on a Bare PC Webserver with Dual NICs," 33rd ACM/SIGAPP Symposium on Applied Computing (SAC), 2018.

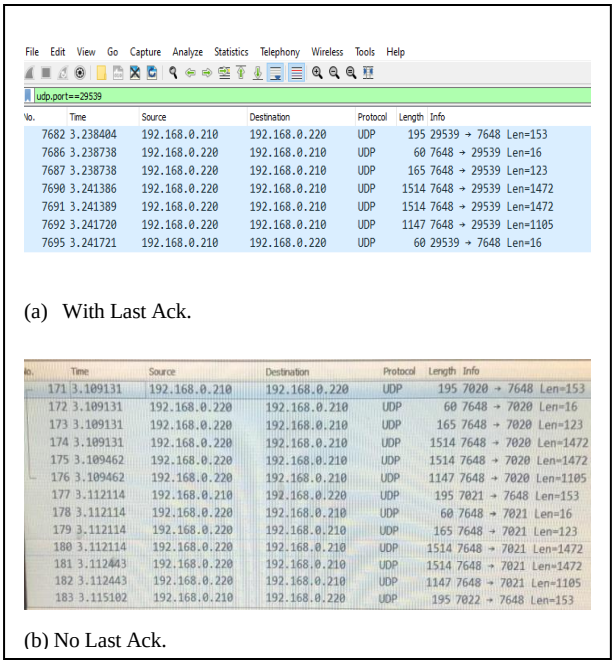

Fig. 4. Protocol packets