A Stateless Bare PC Web Server

Fahad Alotaibi, Ramesh Karne and Alex Wijesinha
Department of Computer and Information Sciences, Towson University, Towson, MD 21252, U.S.A.

Keywords: Bare PC, Bare Machine Computing, Stateless Server, Multi-Core Processor, Web Server, UDP Protocol.

Abstract: Bare PC Web servers that run on 32-bit or 64-bit machines and use TCP or UDP for transport have been built previously. This paper describes the design and implementation of a new stateless UDP-based bare PC multi-core Web server. It also presents performance measurements. The server extends previous server designs with several novel architectural and protocol enhancements. A load balancing technique suitable for multi-core servers is included to illustrate a simple way to efficiently process HTTP requests. The architecture presented here could be adapted in future to build simple conventional Web servers.

1 INTRODUCTION

Conventional Web server designs are complex. Such servers are architected to make the client design simple and the server design complex. Previous approaches to simplify Web server design include use of UDP-based protocols with a last ACK for a 32-bit multi-core Web server (Ordouie et al., 2021), a 32-bit single-core Web server (Soundararajan et al., 2020), and a 64-bit multi-core Web server with a last ACK and no last ACK (Ordouie et al., 2023). When there is no last ACK, all the data packets for an HTTP request are not sent at one time. 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. This approach needs locking and peeking the Ethernet buffers (that is, looking ahead at packets), which results in complex synchronization and load balancing problems. In this paper, the previous designs are extended by designing and implementing a simple stateless UDP-based Web server that handles these problems. Our contributions are as follows.

1. We design and implement a simple and reliable UDP-protocol for HTTP traffic using bare machines.
2. We extend previous work on a multi-core server in (Ordouie et al., 2023) and propose novel architectural and design specifications that enable simple load balancing techniques and avoidance of all locking issues.
3. We conduct experiments to evaluate the performance of the proposed solution in terms of the number of requests, CPU utilization, file size variations, maximum parallel requests, and average processing time.
4. We describe the design of a bare client that works with the proposed stateless server.

The rest of the paper is organized as follows. Section 2 describes the client-server protocol. Section 3 gives an overview of related work. Section 4 discusses architectural and design features of the server. Section 5 provides implementation details. Section 6 presents performance results. Section 7 contains the conclusion.

2 CLIENT/SERVER PROTOCOL

The primary goal of this work is to develop a simple and reliable stateless Web server that uses a UDP-based protocol for transferring HTTP traffic to bare clients over a network. This protocol has no relation to CoAP (Shelby et al., 2014) or QUIC (Iyangar and Thomson, 2021). Figure 1 shows the message exchange for the protocol. The client sends an HTTP GET request to the server for the desired resource file. Upon receiving the HTTP GET request, the server responds with a GET-ACK, which includes important parameters related to the resource file, including file size and the total number of packets to be sent. The client now has all the information needed to complete the transaction.

Alotaibi, F., Karne, R. and Wijesinha, A.
A Stateless Bare PC Web Server.
DOI: 10.5220/0012207400003584
In *Proceedings of the 19th International Conference on Web Information Systems and Technologies (WEBIST 2023)*, pages 406-413
ISBN: 978-989-758-672-9; ISSN: 2184-3252

Note that the client only sends the HTTP GET request and the server only responds with GET-ACK, the HTTP header, and a small number *n* of data packets (for example, *n* may be between 4 and 8). We chose a small value for *n* because there is no automatic retransmission mechanism at the server, the server is stateless, and it is easier for the client to make subsequent GET requests instead of the usual acks for reliability. We use a special 16-byte control header that is sent with each packet (described later) that further simplifies the client and server design.

Figure 2 illustrates how to deal with subsequent packets and lost packets. Here, the client sends a subsequent GET with a starting packet number and the server simply sends the data starting with that packet number. There is no GET-ACK sent for subsequent requests. While this protocol makes the server stateless and simple, it requires a moderate increase in client complexity. This increase is acceptable because most clients have large memory and faster processors, and they are already able to deal with small and large files. One may view the subsequent GETs as replacing the previous reliability mechanisms and out-of-order logic for the client. In this protocol, the client has complete control over its requests in any order using subsequent GET requests. Also, the stateless server has significantly less complexity than a conventional server since it does not need to keep any state about the request. Reliability is now primarily handled at the client side.

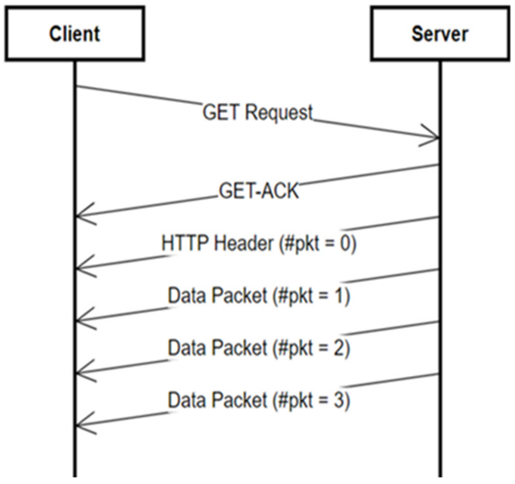

Figure 1: Original GET request.

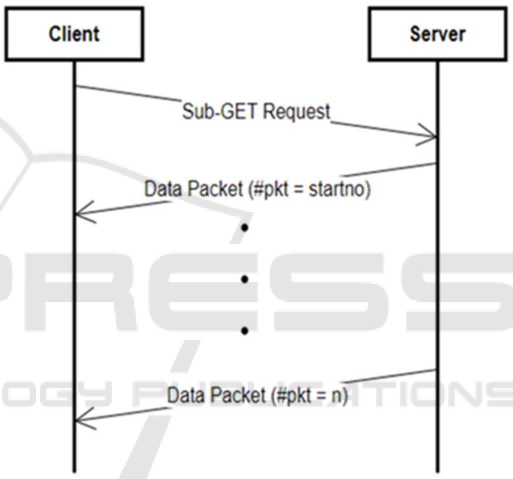

Figure 2: Subsequent GET requests.

3 RELATED WORK

The Bare Machine Computing (BMC) paradigm evolved from the Application-Oriented Architecture (AOA) (Karne, 1995) and Dispersed Operating System Computing (DOSC) (Karne et al., 2005). Previous publications on BMC are at (Karne, n.d.).

Bare applications can run on older or newer x86 and x64 compatible Intel processors. In the BMC approach, a computing device is made bare, meaning that it has no OS and no hard disk, and only uses the BIOS during the boot process. The bare computing device contains no valuable resources such as code, data or applications that need to be protected. The application software is written in C/C++ with a small amount of assembly code to communicate to hardware. Application programs directly communicate with and control the hardware using a hardware API (HAPI). BMC systems are based on a single programming environment and are owner centric. 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. The BMC paradigm eliminates all intermediate layers and middleware enabling applications to be independent of environments. It has been also used to build middleboxes and split servers.

Earlier work on a bare PC Web server for a 32-bit multi-core machine based on TCP (Soundararajan et al., 2020), and UDP (Soundararajan et al., 2020) provide details of similar approaches to build Web servers. Technical details underlying the design of a 64-bit multi-core Web server are given in (Ordouie et al., 2021). Design issues with a 64-bit CPU architecture and multiple cores sharing a single network interface card are discussed in (Ordouie et al., 2023). A preliminary attempt to migrate a 32-bit single core Web server to a 64-bit was made in (Chang et al., 2016). That work used a TCP-based

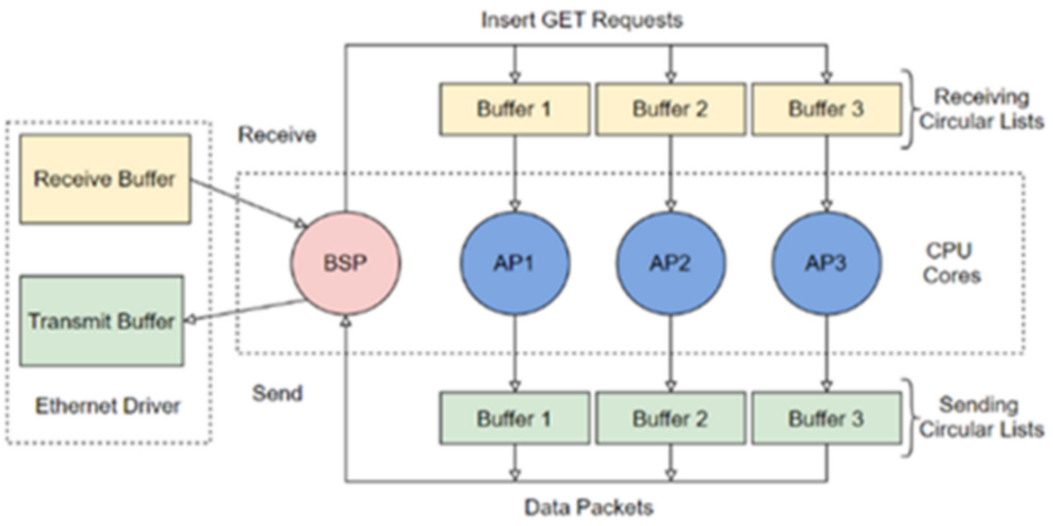

Figure 3: Multi-core server architecture.

server and focused primarily on migration.

Exokernel (Engler, 1998), Microkernel (Odun-Ayo et al., 2021), Tiny-OS (Levis, 2012), and RIOT (Baccelli et al., 2013) are a few examples of approaches to reduce the size and complexity of the OS or kernel, give more direct hardware access to applications, move some OS functions into user space, and bypass the kernel. RDMA is now widely used in the cloud (Kong et al., 2023), and in view of the increased support for kernel bypass in data center servers, (Zhang et al., 2019) propose a library OS architecture for kernel-bypass devices. These approaches differ from BMC in that some form of an OS or kernel is present. Embedded systems that integrate applications with an operating system or kernel, and virtualization approaches are also different from BMC systems.

In a BMC system, conventional OS functions are not duplicated in an application suite as there is no centralized OS or 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 set of applications. The bare-to-bare communication is implemented as application-to-application avoiding all middle layers. There are no heterogeneous components, and the application suite includes the necessary network protocols and device drivers. An application suite image is typically very small since BMC systems are designed to be domain-specific and have only the necessary functionality. For example, the UDP based stateless server executable image size is 331,776 bytes.

4 ARCHITECTURE AND DESIGN

4.1 Architecture

Figure 3 shows the overall system architecture of the bare Web server. The server has four Intel core processors with 4 GB of memory working asynchronously and concurrently to process HTTP requests. We refer to these processors as BSP, AP1, AP2 and AP3, where the BSP is responsible for booting and waking up the other cores. The BSP is also used as a dedicated network processor to send and receive packets over Ethernet. In this approach, the other cores do not send and receive packets directly. Instead, the BSP receives packets and dispatches them to cores AP1, AP2, and AP3 using the round robin algorithm. Alternate approaches such as having all cores peek packets in the Ethernet buffers are inefficient and complex due to concurrency control mechanisms (Ordouie et al., 2023). We note that multi-core architectures typically focus on thread-level parallelism rather than networking. Ethernet bonding may be used with multiple cores to separate receive and send paths using two network interface cards (NICs) (Almansour et al., 2018).

In the BMC paradigm, the programmer has total control over hardware and software, and applications directly communicate to the hardware through a direct hardware API (HAPI). There is no middleware in the bare server. The Ethernet driver is also bare without any OS or kernel support. This architecture is based on a shared memory model wherein all cores have access to main memory. Concurrency control problems are avoided by providing circular lists for input and output packets.

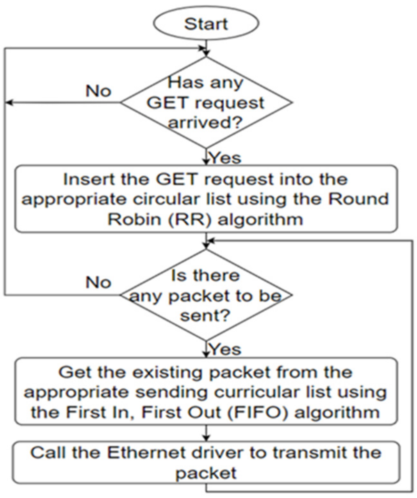
Start
Has any GET request arrived?
No
Yes
Insert the GET request into the appropriate circular list using the Round Robin (RR) algorithm
Is there any packet to be sent?
No
Yes
Get the existing packet from the appropriate sending curricular list using the First In, First Out (FIFO) algorithm
Call the Ethernet driver to transmit the packet

Figure 4: BSP control flow.

When a packet is received by the BSP, it is placed on a given core's input circular list. When there are one or more packets in the input circular list, the corresponding AP processes the GET request by removing them from the circular list. Each AP processes HTTP requests independently without interfering with the other APs. There is thus HTTP request-level parallelism implemented in this design. An AP processes a request without interruption and then places the data packets in the send circular list. All data packets for a given resource file are preformed and kept in memory. Furthermore, each resource file is divided into packets during initialization of the server and pre-processed to be ready to send.

In addition to receiving packets from the Ethernet buffer, the BSP checks the send circular lists for each AP to determine if they have one or more packets to be sent. These packets are inserted into the Ethernet send buffer one at a time in the order they arrived from the APs. As all the cores are running concurrently processing HTTP requests, the single Ethernet card becomes a bottleneck limiting the parallelism that can be achieved. In effect, this bottleneck is the main design issue when implementing Web servers using a multi-core architecture. Concurrency is avoided using the send and receive circular lists for data, and at the Ethernet level by dedicating the BSP to manage receiving and sending packets. Otherwise, the Ethernet receive and transmit buffers must have concurrency control as in (Ordouie et al., 2023). The stateless server architecture presented here is novel, simple, and scalable.

4.2 Processor and Client Control Flow

4.2.1 BSP Control Flow

The control flow shown in Figure 4 illustrates the processing logic for the BSP, which has a loop that consists of receive and send controls. The receive control checks whether a packet is ready to be received from the Ethernet buffers. There are a total of 4096 circular list entries in the bare driver model. The BSP program directly accesses the Ethernet receive buffer entry by checking the DD (device done) bit set and reads the packet into a receive buffer. This packet is allocated to AP1, AP2, or AP3 using round robin by inserting it in the appropriate receive circular lists for the APs. Similarly, send control checks if there are one or more packets in the send circular list. If so, it gets the packet for the list and inserts it into the Ethernet send buffer. As noted earlier, the BMC programmer's code has complete control over the Ethernet driver and related hardware.

4.2.2 AP Control Flow

AP control flow is shown in Figure 5. After an AP has been woken up by the BSP, it remains in a loop to process HTTP requests. It does this by checking if there are one or more packets in the receive circular list and then processing them in order. If there is a GET request to be processed, it calls the RCVCall function. These requests could be either original GETs or subsequent GETs received from the client. The RCVCall function calls the IP Handler, which then calls the UDP handler. At each stage, the appropriate headers are checked and validated.

The UDP handler plays an important role in AP processing. Its logic is different for regular and subsequent GETs. For regular GETs, it needs to send a specific number (*n*) of packets for a given resource file. For subsequent GETs, it must send appropriate packets beginning with a starting number requested by the client. In either case, the UDP handler calls the IP handler with the appropriate number of packets of data to be sent. The IP handler inserts the packets into the corresponding send circular lists for the APs. The above control flows for the APs are executed as a single thread of execution without interruption. As each HTTP request is independent of other requests, this control flow is simple and applicable to all of them.

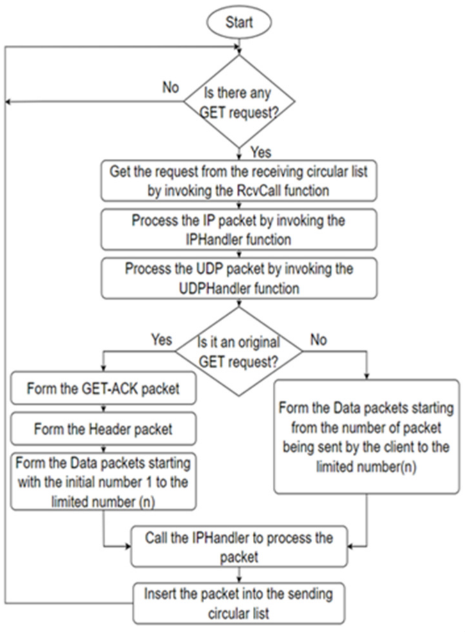
Start
Is there any GET request?
No
Yes
Get the request from the receiving circular list by invoking the RcvCall function
Process the IP packet by invoking the IPHandler function
Process the UDP packet by invoking the UDPHandler function
Is it an original GET request?
Yes
No
Form the GET-ACK packet
Form the Header packet
Form the Data packets starting with the initial number 1 to the limited number (n)
Form the Data packets starting from the number of packet being sent by the client to the limited number(n)
Call the IPHandler to process the packet
Insert the packet into the sending circular list

Figure 5: AP control flow.

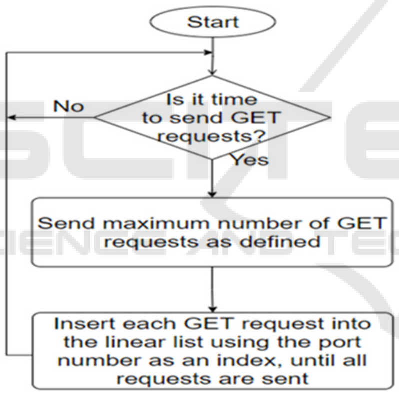
Start
Is it time to send GET requests?
No
Yes
Send maximum number of GET requests as defined
Insert each GET request into the linear list using the port number as an index, until all requests are sent

Figure 6: Client's send control flow.

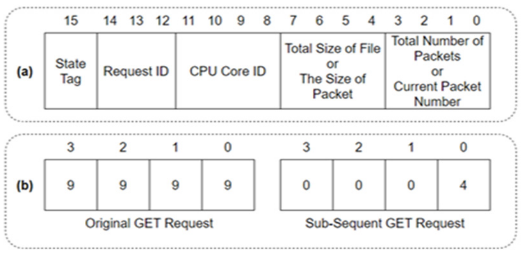

Figure 7: (a) Data header (b) Tail of GET packet.

4.2.3 Client Control Flow

The bare UDP client is used to perform testing and measure performance of the stateless server. Stateless server design impacts the client. The client sends GETs on a periodic basis to the server. This period is characterized by the two parameters frequency and maxreq as shown in Fig. 6. These parameters determine the request rate. For example, if frequency and maxreq values are 9 and 8 respectively, then every 9 units of time, 8 requests will be sent to the server. Each unit of time in our design is 1/4 milliseconds, as defined by the timer period. The 9 units of time amounts to 9/4 = 2.25 milliseconds. Thus, (8/2.25) x 1000 = 3555 requests per second will be sent to the server.

The client logic for received packets is as follows. The client uses port numbers to index requests and maintains the state of requests in a data structure. The state of a request is updated when a new packet arrives and when all data arrivals are complete. Each data packet coming from the server has a data header of 16 bytes. Using this header as control information in the client design makes the logic simpler for implementation. In this header, the packet state inserted by the server indicates the type of packet sent to the client. The client uses the packet state to trigger processing of a given response packet from the server.

There are five states named Get Ack, Header, Data Part, Last Data, and Last Data Now (corresponding respectively to type values 0x31, 0x32, 0x33, 0x34, 0x88). When a packet of type Get Ack, Header, or Data Part arrives, it updates the linear list structures and returns to the caller. When Last Data arrives, it updates the data count and if all data has arrived, it deletes the entry in the linear list. The Last Data Now state indicates that only partial data was sent, which is limited by the number of packets *n* that can be sent at a time for a given request. In this case, the client must send the next GET with a starting packet number to receive subsequent sets of data or for requesting lost packets.

5 IMPLEMENTATION

The server and client are implemented using C/C++ code. All circular lists referred to in the previous section are designed and implemented using C++ classes. A stack structure is used to store request ids for HTTP requests. The stack is initialized with numbers 1–5000 at the start of the program. When a packet arrives in the BSP, a request id is popped from the stack. When a request is complete, the request id is pushed back onto the stack. The maximum number of request ids used indicates the maximum parallelism achievable in the system. Each core also has a core id of 0, 1, 2, 3 (corresponding respectively BSP, AP1, AP2, AP3). The cores AP1, AP2, AP3 are symmetrical. That is, any AP can process any request or any subsequent request. There are many new design features in the stateless server that makes the implementation simple and results in a small code size image as noted before. The 16-byte data header plays an important role in the design. Its details are shown in Fig. 7 (a). In addition, the tail data of a GET request also has some control information which is used at the server and simplifies the server design for subsequent requests. This data has no relation to the HTML data. The state tag value 0x88 (Last Data Now) is used for larger files.

As indicated in the client design, the state field also makes the client implementation simpler. The request number and core id fields helped to test and debug problems at the server related to identifying requests processed by the corresponding cores. The total bytes field is used in Get Ack to indicate the total bytes for a given resource file. The same field is used for packet size in data packets to indicate the current size of the packet. The packet number field is used in Get Ack as the total number of packets for the request. The same field is used as the number of packets when transmitting data packets. In addition, we also added 4 optional characters to GET and subsequent GET requests at the client as shown in Figure 7(b). For initial GET requests the code is 0x9999, and for subsequent GETs the code is a starting packet number. These values are parsed by the server and used in the control logic to simplify the server code.

6 PERFORMANCE RESULTS

The tests were done in a LAN using a gigabit Ethernet switch, a 4-core Dell Optiplex 9010 desktop as the bare server, and four Dell Optiplex 260 desktops as bare clients. We connected 1-4 bare PC clients, where each client can serve up to a maximum of 3555 requests per second. The clients send requests for file sizes of 4K through 128K. The parameter N = 1, 2, 4, 8, 16, 32 is used to vary file sizes from 4K to 128K. The results that follow are based on measurements collected over a 15-minute period. These results are preliminary, and more tests need to be conducted in an Internet environment to validate the design and identify any issues. We have also not considered security issues such as server authentication, and encryption and integrity protection for data packets.

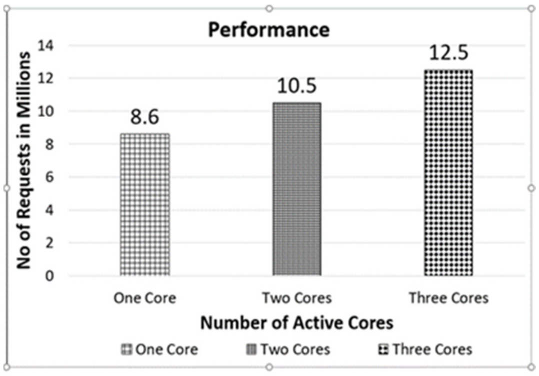

Figure 8: Performance (number of requests).

6.1 Varying the Number of Cores

The graph in Figure 8 shows the number of requests when varying the number of cores (1, 2 or 3) with a fixed file size of 4K. Here, the one core model used three clients (3555, 3555, 2500 requests) with a total of 9610 requests/sec to generate the maximum load, the two-core model used four clients (3555, 3555, 3555, 1000 requests) with a total of 11,665 requests/sec to generate the maximum load, and the three-core model used four clients (3555, 3555, 3555, 3200 requests/sec) with a total of 13,865 requests/sec to generate the maximum load. These numbers show the maximum capacity of the server with stable operation.

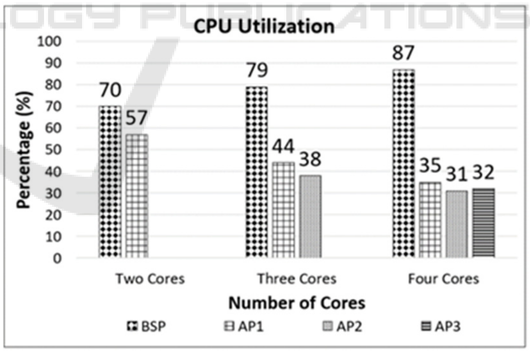

Figure 9: Performance (CPU utilization).

The BSP core is only used for network operations and is not involved in processing HTTP requests. The results indicate that for 1 to 2 cores, the performance increased by 21.5%, and for 1 to 3 cores by 44.7%. This clearly indicates that the speedup is not linear with respect to adding more cores to process HTTP requests. The reason for this low speedup is due to a single network card for multiple cores becoming a bottleneck when processing multiple HTTP requests concurrently. Since HTTP processing is a network-based application, thread-level and application-level parallelism could not be exploited.

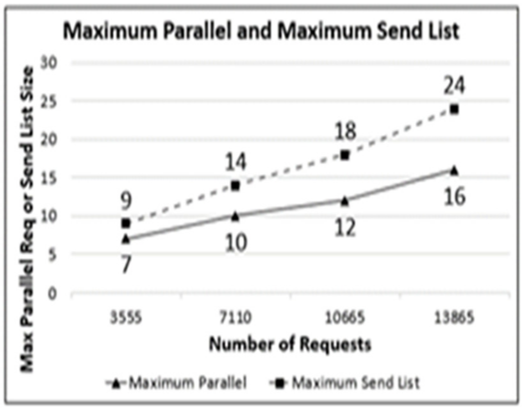

Figure 10: Max parallel and send list size.

Figure 9 shows CPU utilization for a 4K resource file using the preceding measurement parameters. CPU utilization is measured by using the *rdtsc* assembly instruction that gives clock ticks. The clock frequency for the OptiPlex 9010 is 3.4 GHz. The clock tick for this model is (1/ (3.4*10^9)), which is roughly 296 picoseconds. It is seen that BSP utilization reaches 87% with all cores running in the system. Because the APs are not fully utilized in all three models (2, 3 or 4 cores), it limits the speedup in this system. To achieve scalable performance, all cores must be fully utilized. This not possible with one network card as noted above.

6.2 Max Parallel and Send List Size

The BSP receives packets from the Ethernet buffer and distributes them to the other cores. There is one input circular list and one output circular list for each AP. We measured the queue sizes for these lists using the same parameters as above. The maximum number of request ids indicates the maximum number of parallel requests (max parallel) processed at a given time. The input circular list measurements show that there were 1 or 2 requests waiting at a given point.

The cores were free to handle the input circular list without waiting. Max parallel and max send list size were measured when the number of requests is varied in the system by using multiple clients. As seen in Figure 10, max parallel shows a range of 7 to 16, which indicates that at one point there were 16 requests outstanding in the system. Similarly, max send list sizes range from 9 to 24 showing there was a maximum of 24 packets in the send circular list waiting to be sent to the Ethernet. As these numbers are small, they show that the APs and their circular lists were not fully utilized.

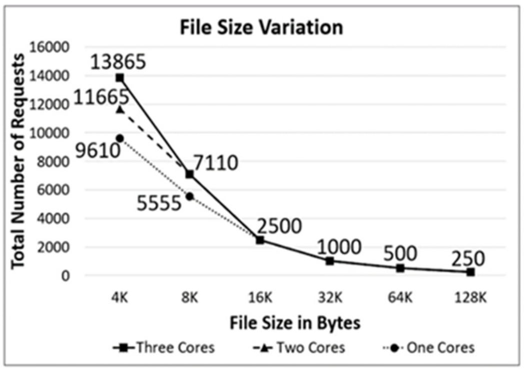

Figure 11: Varying file size.

6.3 Varying the File Size

Figure 11 shows the total number of requests when varying the file size. It is seen that when the resource file size increases, there are more packets and it takes a longer amount of time, thus limiting the number of requests. For the 16K file size, all models (1, 2 or 3 cores) behave the same as they are limited by the network rather than the capacity of the cores. The number of requests has dropped dramatically indicating the limit of this server with a single network card.

6.4 Client Processing Time

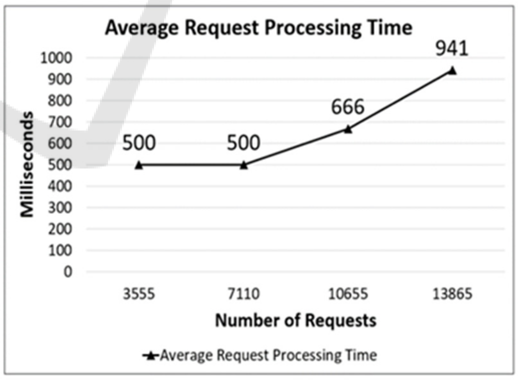

Figure 12: Processing time at client.

Figure 12 shows the average processing time on the client side when the load is varied at the server using multiple clients. The average processing time at clients varies between 500 to 941 milliseconds. However, the average processing time for requests measured at the server is only 27 milliseconds. This is because at the server, we only measured the processing time until the packets were inserted into the send buffer, which does not include actual transmission at the Ethernet level. The average processing time for requests at the client reflects the actual processing time and it shows how the server performs with an increased load.

7 CONCLUSION

We described the design and implementation of a novel stateless 64-bit multi-core Web server that runs on a bare machine. One core handles networking while other cores process the HTTP requests. The server communicates with bare machine clients using a simple UDP-based protocol that is easy to implement. We also gave a brief overview of the client design.

A key aspect of the protocol is the use of a 16-byte data control header with fields specifically designed to simplify client-server communication. The server architecture avoids concurrency controls by using buffers at the receiving and sending ends. The receive circular list did not affect the results. The send circular list showed a varying number of packets (maximum of 24) waiting to be sent depending on the server load. The measured concurrency in the system shows reasonable parallelism (maximum of 16). The use of a dedicated core for networking enables multiple cores to be used efficiently to implement the Web server application.

We identified the single network interface card as the main bottleneck in processing requests in multi-core processors. The performance measurements indicate that there is no linear speedup gained by using multiple cores for processing because the network interface is the bottleneck. Future studies could investigate the use of multiple on-board NIC interfaces or chips for multi-core processors.

REFERENCES

Ordouie, N., Soundararajan, N., Karne, R., and Wijesinha, A. L. (2021). Developing Computer Applications without any OS or Kernel in a Multi-core Architecture. International Symposium on Networks, Computers and Communications (ISNCC).

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

Ordouie, N., Karne, R., Wijesinha, A. and Soundararajan, N. (2023). A Simple UDP-Based Web Server on a Bare PC with 64-bit Multi-core Processors: Design and Implementation. 2023 International Conference on Computing, Networking and Communications (ICNC).

Karne, R. K. (1995). Object-oriented Computer Architectures for New Generation of Applications. Computer Architecture News, Vol. 23, No. 5.

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

Karne, R. (n.d.). Bare Machine Computing, http://orion.towson.edu/~karne/dosc/pubs.htm. Accessed July 3, 2023.

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

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

Engler, D. R. (1998). The Exokernel Operating System Architecture. Ph.D. thesis, MIT.

Odun-Ayo, I., Okokpujie, K., Akinwumi, H., Juwe, J., Otunuya, H., and Oladapo, A. (2021). An Overview of Microkernel Based Operating Systems. IOP Conference Series: Materials Science and Engineering, 1107 012052, 2021.

Levis, P., (2012). Experiences from a Decade of TinyOS Development. 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI '12).

Baccelli, E., Hahm, O. Gunes, M., Wahlisch, M., and Schmidt, T. C. (2013). RIOT OS: Towards an OS for the Internet of Things. 2013 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS).

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

Shelby, Z., Hartke, K., and Bormann, C. (2014). The Constrained Application Protocol (CoAP). RFC 7252.

Iyengar, J., and Thomson, M. (2021). QUIC: A UDP-Based Multiplexed and Secure Transport, RFC 9000.

Kong, X., Chen, J., Bai, W., Xu, Y., Elhaddad, M., Raindel, S., Padhye, J., Lebeck, A. R., and Zhuo, D. (2023). Understanding RDMA Microarchitecture Resources for Performance Isolation. 20th USENIX Symposium on Networked Systems Design and Implementation.

Zhang, I. Liu, J., Austin, A., Roberts, M. L., and Badam, A. (2019). I'm Not Dead Yet! The Role of the Operating System in a Kernel-Bypass Era. 17th Workshop on Hot Topics in Operating Systems (HotOS '19).