A Novel Client/Server Protocol for Web-based Communication over UDP on a Bare Machine

Nirmala Soundararajan
Computer and Information Sciences
Towson University
Towson, MD USA
nsoundararajan@towson.edu

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

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

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

Bharat S Rawal
Computer and Information Sciences
Gannon University
Erie, PA USA
rawalksh001@gannon.edu

Abstract*— Bare machines enable clients and servers to communicate without using an operating system or kernel. We describe the design and implementation of a novel UDP-based protocol for communication between a bare machine Web server and client using HTTP. The protocol is simple, easy to code, reliable and inherently secure. We include code snippets to illustrate unique features in our approach. Results from preliminary measurements are also given to demonstrate the potential performance improvement due to using UDP instead of TCP.

Keywords—bare machine computing, bare PC, UDP, client/server communication, Web server, Web client

I. INTRODUCTION

Bare machine computing (BMC) has two distinguishing characteristics. First, the computers are bare without any hard disk or resident OS or kernel. Second, the programming paradigm is different since a given application suite and its programs directly communicate with the hardware. BMC applications and their system aspects are integrated to run as a single program. For example, all the network protocol and device driver code are integrated with the BMC application since there is no separation of user space and kernel space in a bare machine. BMC systems thus provide a simple OS-free environment to test novel network protocol designs.

Most applications today use TCP connections for client/server communication. TCP provides reliability (via connection establishment and closing, sequence and ack numbers, re-transmissions, and transmission timeout) flow control (via sliding window and window advertisements); and congestion control (via additive increase, multiplicative decrease, and slow start). For security, TCP connections use TLS. However, TCP introduces additional overhead, poses numerous complexities, results in a large amount of code, and makes debugging difficult. Previously, a TCP-based bare machine Web server was built to run on a multicore architecture [1]. Motivated by TCP-related design issues identified in that work, we simplify HTTP-based client-server communication between bare machines by using UDP instead of TCP. While the QUIC transport protocol uses UDP to provide similar services to those offered by TCP and TLS to applications and higher layer protocols such as HTTP/3 [2], HTTP over UDP on a bare machine is a customized minimal design that has no relation to QUIC.

BMC systems have some similarities to systems for application-level hardware resource management [3], library OSs [4], OSs for embedded wireless sensors [5], high performance OSs for virtualized environments [6], systems for improving buffering and caching performance [7], toolkits for building OSs [8], scaled-down Linux systems that run directly on the hardware [9], OSs for low-power devices [10], and multicore architectures designed for high performance [11].

Many BMC applications have been previously built. Examples include a Web server with Ethernet bonding [12], a TLS mail server [13], a peer-to-peer VoIP system [14], a mail server with SQLite [15], servers that split the TCP protocol for high performance [16], a text-based browser [17], and a Web server for a multicore system [18].

BMC applications are multi-threaded enabling them to perform well. Additionally, there are no vulnerabilities due to an OS. The programs are small in size and only features essential to a given application are supported. There is no external software needed and the user controls the BMC application.

The rest of this paper is organized as follows. Section II describes the integrated TCP and HTTP protocols as usually implemented on a bare machine. Section III describes the integration of UDP with HTTP. Section IV presents the server and client design. Protocol implementation is detailed in Section V, testing and evaluation results are given in Section VI, and Section VII contains the conclusion.

II. INTEGRATING TCP AND HTTP

The original bare machine Web server runs HTTP over TCP, where internally the protocols are integrated with the application as shown in Fig.1 to provide communication between client and server. As the BMC paradigm has no layering, integrating all the

necessary modules including IP and Ethernet protocol is easier than in a typical conventional system. Notice that the three phases in the TCP protocol (connection, data transfer and termination) are as in a conventional system.

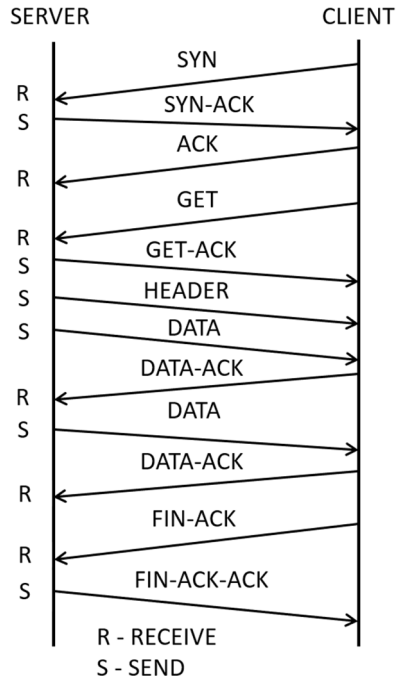

Fig. 1. TCP over HTTP on a bare machine.

The SYN, SYN-ACK and ACK messages are used to establish the connection as usual. An abbreviated FIN-ACK and FIN-ACK-ACK exchange is used to close the connection and speed up the process. A bare machine TCP-based Web server creates a TCP state table (TST) with a unique entry for each client's request based on a client's IP address and port number. The interactions between a client and the server result in state transitions that are captured in the TST table and help to complete a client request. Reliability of the data transfer is achieved by providing retransmissions and keeping track of sequence numbers and acknowledgements.

Each client request also requires a new thread to track its state transitions. In the bare machine Web server application, a process model is used as shown in Fig. 2. There are three types of tasks in this system. The Main task runs all the time in the application. When a packet arrives, a Receive task runs to process the request and carries the single thread of execution through Ethernet, IP, TCP, and HTTP, and updates the TST. Control returns to the Main task after the packet is processed. When a GET (HTTP request) arrives, a separate HTTP task is created for each request and it runs until the connection is terminated. However, this task goes through many state transitions due to TCP and results in a complex implementation. The size of the TCP code in a BMC implementation using C++ is 5800 lines not including header files, and Ethernet and IP processing. The bare machine Web client is built by using the same Web server design as above, but by reversing the roles. This Web client does not have any graphics capabilities as its main purpose was to stress test the bare machine Web server.

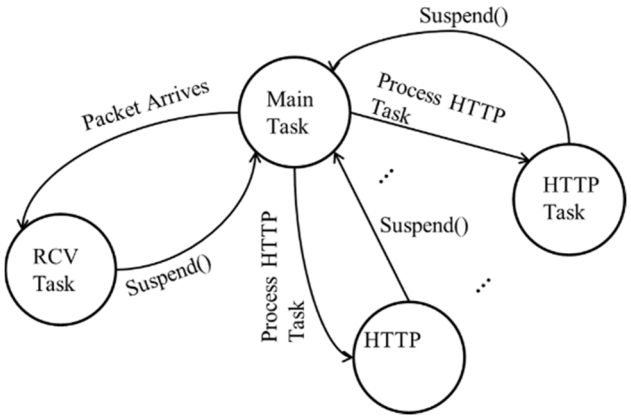

Fig. 2. Process model using TCP.

III. Integrating UDP and HTTP

Fig. 3 shows the message exchange for the integrated UDP and HTTP protocols on a bare machine. The integrated protocols enable the client and server to communicate at the application level. By providing reliability at the application level via UDP, the complexity due to TCP is avoided. While this application is implemented at present on a bare machine, it can be adapted to run on conventional systems at the cost of incurring more overhead due to the OS. The basic design features in this application could also be used in future for implementing the QUIC protocol on a bare machine.

Once the server receives the GET request from a client, it sends a GET-ACK which includes a data header and ack. The data header consists of 16 bytes as shown in Fig. 4. This data header consists of a request id, CPU (core) id, size, and the number of packets. The 16 bytes are part of the UDP data. The request id is unique and generated by the server to track requests received from a client. The CPU id is a unique id for a given processor or a core, which is relevant for multicore implementations. At present, this field indicates the core that is servicing a given packet. It can also be used for other purposes such as load balancing or failure reporting of a core. The size field indicates the total size of the requested file. The #pkts field shows the number of packets that will be transmitted by the server to a client. After sending the GET-ACK, the server sends the requested data, one packet at a time. Each data packet also carries a data header in addition to data. The size field in this data header indicates the size of a given packet and the #pkts field indicates the number of the packet that is being transmitted. The client also sends data acks with a header and an ack. In this case, the header field size indicates how much data has been received and #pkts indicates the total packets received. After a client receives all packets and data, it will send a last ack, which also has a header and an ack. The size field in this ack indicates the total size of data received by the client and #pkts indicates the number of packets received.

The data header has a request id and CPU id fields in addition to the other fields. The client does not add any header field when it sends a GET request. All other messages carry a data header. The server header fields request id and cpu id in the GET-ACK are saved at the client for each request. These two fields must be returned by the client in every message sent to the

server. When a packet is received at the server, these fields are checked. If they do not match, the server will reset the client due to the error (this is not shown in Fig. 3). Note that the size and #pkts fields in the data header have a different meaning depending on whether a packet is sent by the server or by the client. By making a few modifications, the communication could be secured by adding mechanisms for server authentication, data encryption and data integrity.

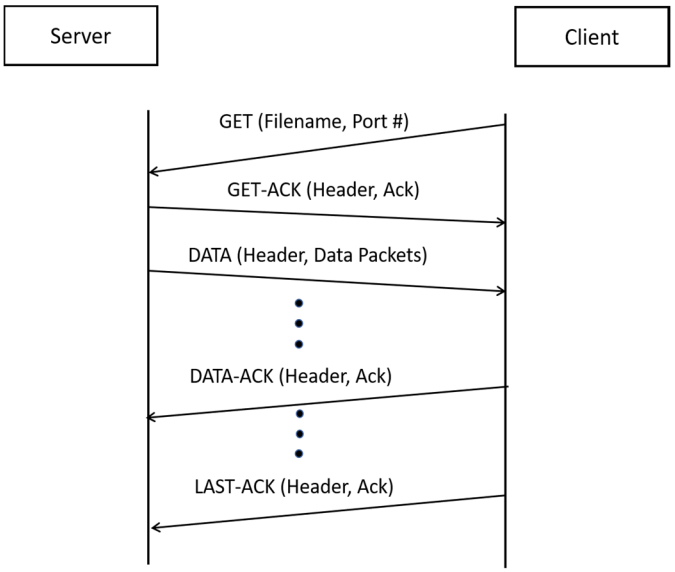

Fig. 3. UDP/HTTP client server interfaces.

Req id – A unique id at the server to distinguish Client Requests
CPU Id – A processor Identification
Size – Total Size of a packet / Data received thus far
Pkts – Total number of packets / Packets received

Fig. 4. Data header format.

IV. Design

By using a UDP-based protocol to support HTTP data exchange on a bare machine, many aspects of the design are simplified. In addition, the implementation is simpler, and it is much easier to debug the application since there is less code. Some key protocol design details for the client and server are given below. The significant reduction in code size as compared to the TCP implementation in [1] is primarily due to the simplicity of the new design.

A. Server Design

For a given Web site, there can be several locally resident files that may be static or dynamic. This paper only focusses on static file handling.

a) Resource Files: For a given Web server, its resource files are known in advance. This implies that the file sizes and packet lengths in the Ethernet packets are also known. Using this information, we divided resource files into packets based on the packet size limits (e.g. 1460 bytes) and calculated the number of packets needed for each resource file. As bare machines use real memory, the resource file addresses are also known prior to sending data to a client. This information along with other relevant details are stored at the server. When the client's GET request is received, the server forms the data header (this is possible because the values for size and #pkts are known). Since there are no sequence numbers and ack numbers in this design, the preformed data packets are ready to be sent except for the request id, which is inserted in the packet for a given request before sending the packet. This design enables the complexity in the TCP-based design to be avoided and makes the coding simpler. It also improves the performance.

b) Data Structures: Fig. 5 shows the client record needed for storing the state of a client request in the server. All the fields shown here are used to track the state of the request, the client parameters and the data received from the client in the GET request. Since indexing is used, it is also not necessary to search for a request. As shown in Fig. 6, a client's list is defined with the size n (e.g. 4000), where the list consists of 0 to n-1 client records. The index for this list is based on the request id. Thus, n request id's are stored in the bare machine stack during initialization. When a new client request arrives, the stack is popped, providing the request id for a given request. When the request is completed, the request id is pushed on the stack. When the stack becomes empty, the capacity of the server is reached. In this paper, we do not discuss approaches to extend capacity limits (which can be based on CPU utilization and load balancing strategies).

c) Process Model: The process model is much simpler than in Fig. 2 since there is no need to create an individual HTTP task for a given request. In the new UDP-based protocol design, the Main task does all the work and Rcv task is replaced by a Rcv Call() function.

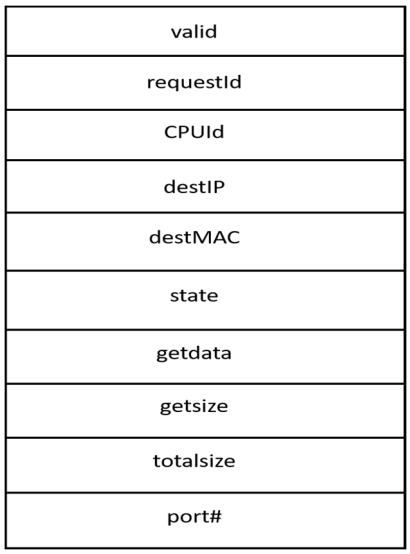

Fig. 5. Client record.

d) Peeking the Ethernet Buffers: In a bare machine, the Ethernet driver is part of the application, and full access to the Ethernet driver API is available at the application level. There are two circular lists in the device driver, consisting of 4096 or 8192 descriptors for the gigabit Intel GB Ethernet controller or card. Each descriptor has status and control information consisting of 16 bytes. The communication between hardware and software is achieved through these descriptors. The status bit in the descriptor indicates whether a packet arrived or has been transmitted, and peeking of this bit is done in the Main task. If the incoming packet is for a GET request, the client's stack is popped for the request id. For other requests, the request id is in the data header. When a packet arrives, the Main task calls Rcv Call() as noted earlier.

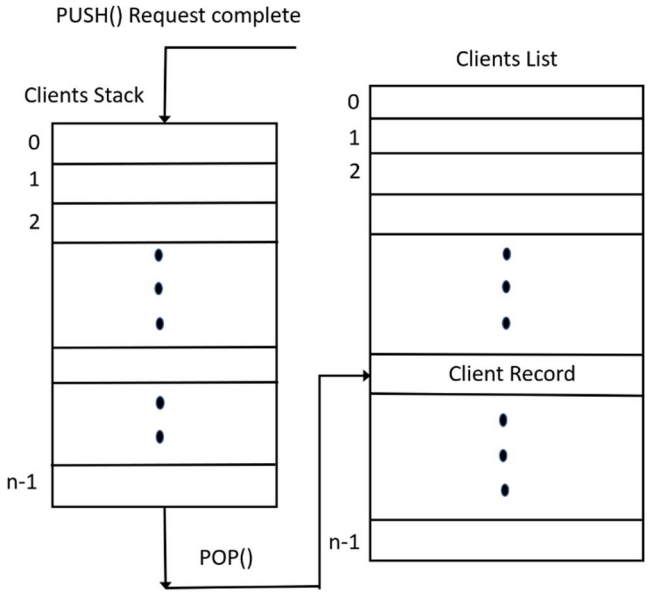

Fig. 6. Client's list and Stack.

e) Control Flow: Fig. 7 shows the control flow. From the Main task, the RDescDone bit is checked in the RDL which signals packet arrival in the Ethernet receive buffer. The RcvCall() function is then called and appropriate actions are taken depending on the type of the packet, which is given by the rtype value. When a GET request is received, a new request id is generated and the status is updated in the client's list. Next, the IP Handler and UDP handler are called to process the IP and UDP headers in the packet. When the GETACK and the Data are sent back to the client, the status in the server is updated. If the packet arrived is ACK, then the Client's list is updated accordingly. If the packet is LASTACK, it signals the end of the request, and the request is removed in the client's list.

B. *Client Design*

The design of the original bare machine TCP Web client is modified to obtain the corresponding UDP client. The data structures used are the same as for the server design detailed in Section A above. A GET request is formed with the desired request rate (in requests/sec) and the client's list is updated. There is no peeking done in the client as it depends on the server to send packets and processes them as they come in. Fig. 8 shows

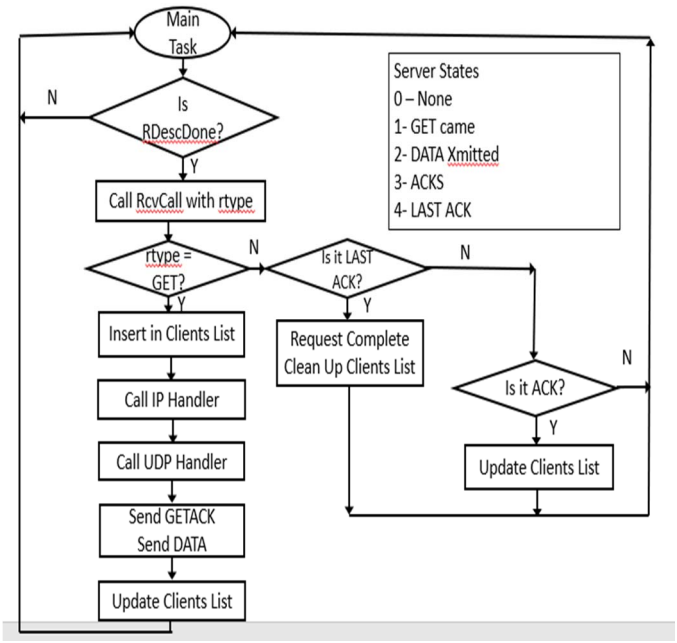

Fig. 7. Server control flow.

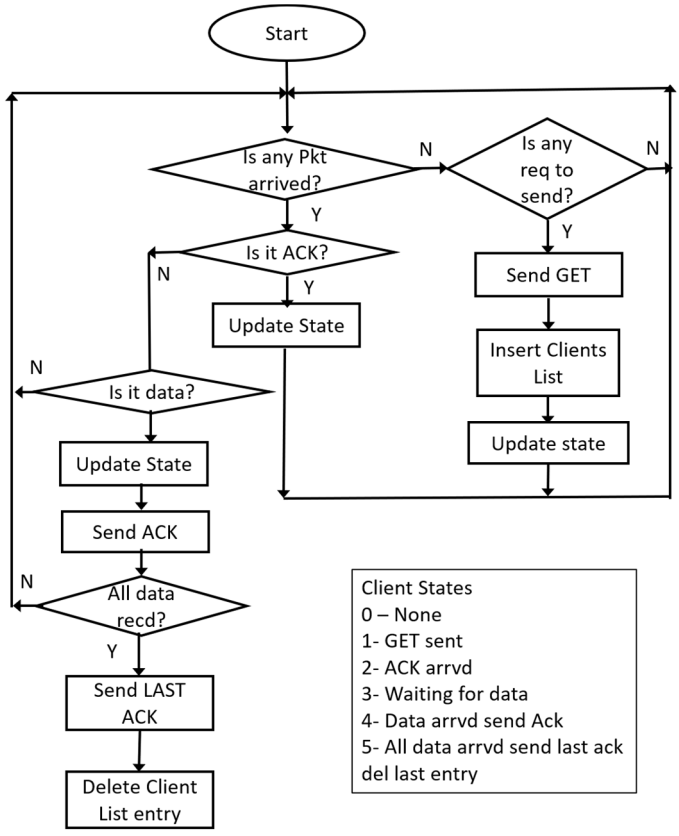

Fig. 8 Client control flow.

the control flow for the client. The client sends the GET request and updates its state and the client's list. After sending the GET request, it waits for the GETACK and data to arrive. When all

the data has arrived, it sends the LASTACK indicating that it has received all the data and finally deletes its client's list entry.

V. Implementation

The Web server and Web client were implemented using C++ and a small piece of code in assembly. The code sizes of important components are shown in Table I. A code snippet is shown in Fig. 9. The Ethernet driver object EO is called to peek into the status bit and check for an arrived packet.

A. Server Implementation

This section describes the control flow of the implementation in step-by-step function calls. The control flow-chart shown in Fig. 7 provides a high-level view of this implementation. The first step is to peek the Ethernet buffer status to check if a packet arrived and set the r-type. If a packet is received, it is copied into the client's list. This method requires access to the Ethernet receive buffer and peeking the the status bit. If peeking is successful, the OUT pointer for the Ethernet buffer circular list is incremented.

TABLE I. Sizes of Important Code Components.

Component	Code Size
Server.exe	268 KB
Client .exe	114 KB
MainTask	409 lines
RcvCall()	175 lines
IPObj	564 lines
UDPObj	449 lines
EtherObjRcv	2876 lines
EtherObjSend	2610 Lines

The code flow shows the sequence of actions after peeking the packet status. The code illustrates the simplicity of the bare machine design and the ability of the application to directly communicate with the hardware. Except for the application, there is no additional software or any form of middleware or OS running in the bare machine.

B. Client Implementation

The client implementation is very similar to that of the server. It is also written in C++ using the direct bare machine hardware interfaces. The client code does not currently have any graphics i.e. the user interface is text only. In future, the text-only browser [17] can be enhanced to include a graphical interface and integrated with this client implementation.

VI. Testing and Evaluation

The implementation was tested with a gigabit Ethernet switch and a pair of Dell Optiplex 9010 computers to run the BMC Web server and client in a laboratory environment. The packets sent during communication between the client and server

```
rtype = EO.isRDescDone(bcoreid, takeget, &pktsize, &preqid);
//the above function peeks for status bit, if packet arrived,
// it copies the data into a buffer, it does stk.pop(&reqid)
//and increments the circular list OUTPTR
    if (rtype == 1 || rtype == 2)
      // Rcv Call function
     RcvCall(preqid, pktsize, rtype);
   retcode = ip.IPHandler(reqid, dptr, pktsize-14, macaddr, starttime, rtype);
    //IPHandler calls
  retcode=udp.UDPHandler(reqid, &IPPack[HeaderLength],(TotalLength-HeaderLength),&IPPack[12],&IPPack[16],protocol, macaddr, currenttask);
//UDPHandler calls
    retcode = processUDPRequest(reqid, &UDPPack[8], DestPort, Size, SourceIP, dmac, currenttask);
//processUDP Request has the following logic.
    if (UDPPack[0] == 'G' && UDPPack[1] == 'E' && UDPPack[2] == 'T' && rtype == 1)
    {
     if (llist0.full() == 0 && stk.empty() == 0) //stack has the indexes
 {
      retcode = llist0.insertGET (reqid, coreid, dip, dmac, dport, UDPPack, psize);
     retcode = processHTTPRequest (reqid, coreid, UDPPack, dport, psize, dip, dmac);
}
      else
            {
              //error
            }
 }
    else if (rtype == 2)
     {
       //we get ACK or LAST ACK from the client
       io.AOAExtractDataHeader(UDPPack,&reqid,
       &coreid, &totsize, &nopkts);
       if ((totsize == llist0.getTotSize(reqid)) && dport ==
            llist0.getDport(reqid))
        {
          llist0.deleteEntry(reqid);
          if (stk.full() == 0)
               stk.push(reqid);
```

Fig. 9. Code snippets for Ethernet driver.

captured by Wireshark (using an additional Windows PC) are shown in Fig. 10.

The TCP-based implementation [1] achieved 8400 requests/sec in a multicore architecture with 4 cores for the server and 3 clients using a resource file size of 4000 bytes. For the same resource file with the same configuration, the request rate for the UDP-based implementation using a single core was 13000 requests/sec. The time taken for a typical request using

UDP was 0.323 milliseconds as seen in Fig. 10 compared to more than 4 milliseconds using TCP.

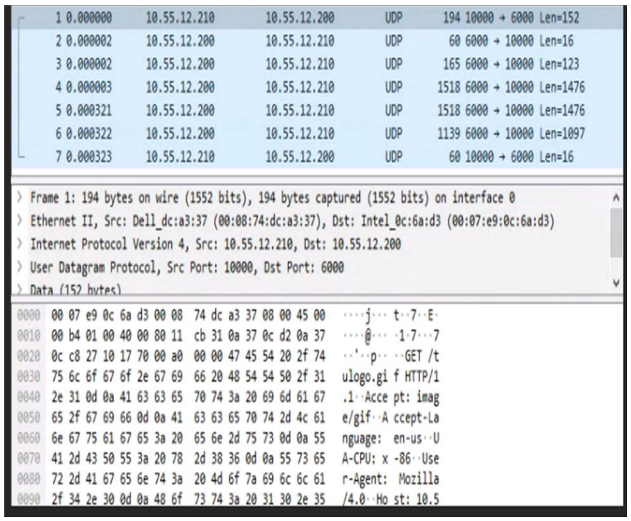

Fig. 10. Packets exchanged by client and server.

VII. CONCLUSION

We described a simple protocol for communication between a bare machine Web server and client that use HTTP over UDP instead of TCP. The HTTP and UDP protocols are integrated in the application. Reliability of data transfer was guaranteed without incurring TCP overhead at the protocol level. Interesting design features include peeking Ethernet buffers, indexing-based data structures, ability of the application to directly communicate with the hardware, simplicity, small code sizes, and elimination of OS-related security vulnerabilities. A similar approach can be used to convert other TCP-based bare machine applications to run on UDP. A similar approach can also be used to implement protocols such as QUIC without any OS or kernel running in the machine. Future work includes extending this design to run on multiple cores and investigating issues related to concurrency control and shared memory multiprocessing.

REFERENCES

[1] N.Soundararajan, H.Chang, R.K.Karne, A.Wijesinha, N.Ordouie, "Design Issues in Running a Web Server on Bare PC Multicore Architecture", 2020 IEEE 44th Annual Computers, Software and Applications Conference, (COMPSAC) July 13th – 17th 2020 virtual event

[2] M. Bishop, Ed., "Hypertext Transfer Protoc0l Version 3 (HTTP/3) draft-ietf-quic-http-29." June 2020.

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

[4] G.Ammons et. al, "Libra: A Library Operating System for a JVM in aVirtualized Execution Environment", VEE '07: Proceedings of the 3rd international conference on Virtual execution environments, June 2007.

[5] TinyOS is an open-source operating system designed for wireless embedded sensor networks http://www.tinyos.net/.

[6] J. Lange, K. Pedretti, T. Hudson, P. Dinda, Z. Cui, L. Xia, P. Bridges, A. Gocke, S. Jaconette, M. Levenhagen, R. Brightwell, Palacios and Kitten: "New High Performance Operating Systems for Scalable Virtualized and Native Supercomputing", Proceedings of the 24th IEEE International Parallel and Distributed Processing Symposium (IPDPS), April 2010.

[7] V. S. Pai, P. Druschel and W. Zwaenepoel. IO-Lite: "A Unified I/O Buffering and Caching System". In Proc. ACM Transactions on Computer Systems, Volume 18, Issue 1, pp. 37 – 66. February 2000.

[8] http://www.cs.utah.edu/flux/oskit. The OS Kit Project.

[9] 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, No. 2, pp. 319-329, 2005.

[10] RIOT: "The friendly Operating System for the Internet of Things", https://www.riot-os.org

[11] Raw architecture Workstation, http://groups.csail.mit.edu/cag/raw/purpose/.

[12] F. Almansour, R. Kartne, A. Wijesinha, and B. Rawal, "Ethernet Bonding on a bare PC Web Server with dual NICs", 33rd Annual ACM Symposium on Applied Computing (SAC), pp. 1116-1121, April 2018.

[13] P. Appiah-Kubi, R. K. Karne, and A. L. Wijesinha. "A Bare PC TLS Webmail Server", The International Conference on Computing, Networking and Communications (ICNC), , pp. 149-153.

[14] G. H. Khaksari, A. L. Wijesinha, and R. Karne. "Secure VoIP using a Bare PC" 3rd International Conference on New Technologies, Mobility and Security (NTMS), 2009.

[15] H.Alabsi, R. K. Karne, A. Wijesinha, R. Almajed, B. Rawal, and F. Almansour, "A Novel SQLite-Based Bare PC Email Server", 15nth International Conference, BDAS2019, Ustron, Poland, May 28-31, 2019, p341-p353

[16] B. S. Rawal, R. K. Karne, A. L. Wijesinha. "Split Protocol Client Server Architecture", Seventeenth IEEE Symposium on Computers and Communications (ISCC'12), July 1 -4, 2012, Cappadocia, Turkey.

[17] S.Almutairi, R. K. Karne and A.L. Wijesinha, "A Bare PC Text Based Browser", 2019 Workshop On Computing, Networking and Communications (CNC), Honolulu, Hawaii, February 2019.

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