Volume 3, No. 4, July- August 2012

ISSN No. 0976-5697

International Journal of Advanced Research in Computer Science

RESEARCH PAPER

Available Online at www.ijarcs.info

QuicX: A light weight protocol for effective communication in Linux Clusters

Pushpendra Gupta*
Department Of Information Technology
Pune Institute of Computer Technology
Pune, India
pgupta.pict@gmail.com

Ankit Agarwal
Department Of Information Technology
Pune Institute of Computer Technology
Pune, India
ankitagarwal.pict@gmail.com

Vivekanand Adam
Department Of Information Technology
Pune Institute of Computer Technology
Pune, India
Vivekadam21@gmail.com

Abstract: A protocol for efficient communication on clusters of Personal Computers (PCs) using the Linux operating system is proposed. In high performance computing (HPC) cluster systems, the physical transmission time is small compared to the time required to process the TCP/IP protocol stack. In this way protocols such as TCP/IP causes an overhead that represent an important amount of communication cost. The headers added at each layer are insignificant in closed network. Our Light weight protocol can be used for efficient communication in clusters using the Linux operating system. It uses an approach to optimize the communication performance in a cluster of computers different from that of providing a user-level interface that removes the operating system (OS) mediation in the communication path. Thus it reduces memory latencies and increases the bandwidth figures. It provides an optimized OS support to reliable and efficient network software that avoids the TCP/IP protocol stack and unnecessary buffer copies.

Keywords: TCP/IP Protocol, Clusters, Linux OS, Communication Protocol, Buffer Copies.

I. INTRODUCTION

Although network bandwidths are increasing and network latencies are decreasing, it is not easy for applications to take advantage of these performance improvements due to the overheads imposed by the many layers of software required for communication. The main approaches adopted to reduce this software overhead have been the improvement of the TCP/IP layers, and the substitution of the TCP/IP layers by alternative layers [2][4][12].

Present system uses TCP/IP protocol for data exchange which includes a lot of processing overhead [1]. Firstly, a number of copies are created during exchange and secondly a header is added at each layer as data passes through various layers of TCP network stack. A single transfer involves 4 buffer copies [Figure 2] that is Read Buffer, Application Buffer, Socket Buffer and NIC buffer and a header (54 – 134 bytes)[Figure 1] including TCP header(20 - 60 bytes), IP Header(20 – 60 bytes) and Ethernet header(14 bytes) added to each packet. In a cluster system, these headers are not required at all. Since the nodes are connected in a closed network there is no need of IP addresses for identifying the nodes. Instead MAC address present in Ethernet header is sufficient enough to identify the destination node. Thus the IP field is unnecessary. Similarly a number of fields are not required which can be eliminated and thus processing time can be saved.

The Buffer copies involve copying of data between kernel context and user context [10]. These copies are generated in TCP/IP communication which increases latency. These copies are again a great overhead.

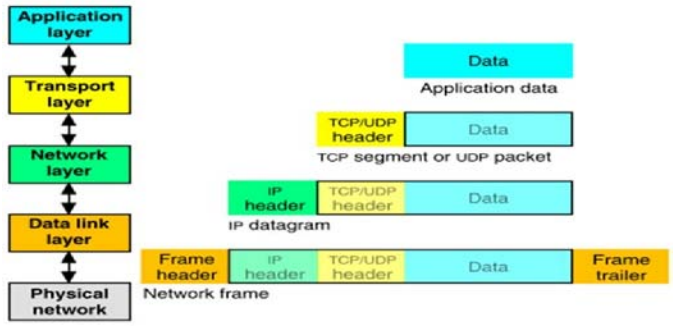

Figure 1. Header Addition

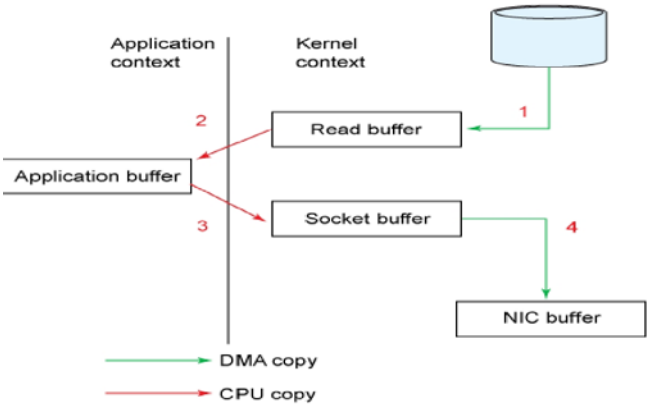

Figure 2. Buffer Copies.

II. DETAILED DESIGN

Our system will consist of a protocol designed to work in place of TCP/IP layers. We will be using the concept of raw sockets [8] for designing our light weight protocol. In computer networking, raw socket is a socket that allows direct sending and receiving of network packet by applications[5], bypassing all encapsulation in the networking software of the operating system. Rather than going through the normal layers of encapsulation/decapsulation that the TCP/IP stack of the kernel does, we will just create the entire packet, add any headers required for proper transmission and will directly pass the packet to the Ethernet layer for transmission[3]. Thus the entire kernel network stack will be bypassed.

The Design supports the following features:

A. Reduced Buffer Copies:

The concept of memory mapping [6] can be used to eliminate almost 3 buffer copies. Memory mapping is a concept that allows an application to map a file into memory, meaning that there is a one-to-one correspondence between a memory address and a word in file. The Programmer can then access the file directly through memory, identically to any chunk of memory resident data – it is even possible to allow writes to the memory region to transparently map back to the file in disk. Linux implements memory mapping by using mmap() system call [3][6] for mapping objects into memory. Memory mapping from user level code is done with a call like

setsockopt(fd, SOL_PACKET, PACKET_RX_RING, (void *) &req, sizeof(req))

The most significant argument in the previous call is the req parameter, this parameter must to have the following structure:

```
struct tpacket_req
{
unsigned int tp_block_size;      /*Minimal size of
                contiguous block */
unsigned int  tp_block_nr;       /* Number of blocks */
unsigned int  tp_frame_size;     /* Size of frame */
unsigned int  tp_frame_nr;       /* Total number of
                frames */
};
```

This structure is defined in /usr/include/linux/if_packet.h and establishes a circular buffer (ring) of unswappable memory mapped in the capture process. Being mapped in the capture process allows reading the captured frames and related meta-information like timestamps without requiring a system call.

Captured frames are grouped in blocks. Each block is a physically contiguous region of memory and holds tp_block_size/tp_frame_size frames. The total number of blocks is tp_block_nr. Note that tp_frame_nr is a redundant parameter because

frames_per_block = tp_block_size/tp_frame_size

A frame can be of any size with the only condition it can fit in a block. A block can only hold an integer number of frames, or in other words, a frame cannot be spawned across two blocks, so there are some details you have to take into account when choosing the frame size.

The mmaped memory is in the form of a circular ring. The memory is divided into blocks of fixed size (here 4096 bytes). Each block is further divided into 2 frames of 2048 bytes. Each frame stores data along with sufficient information about the packet including packet timestamp and flags to indicate whether the frame is ready for sending or free. The Structure is as shown:

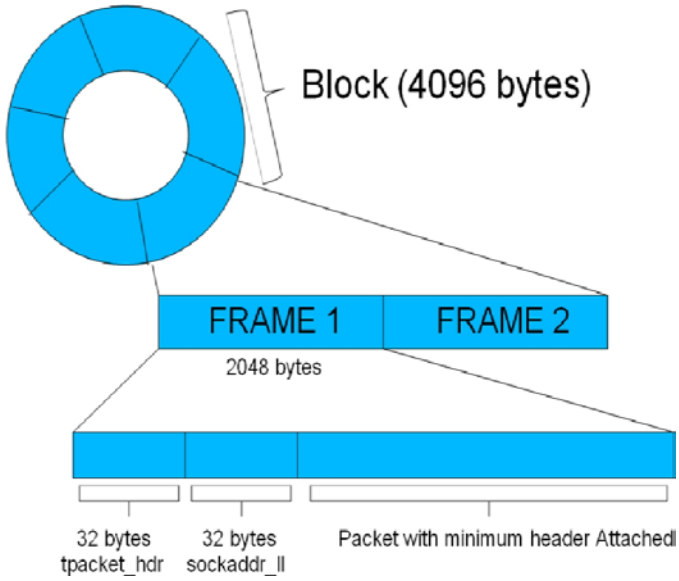

Figure 3. Mmaped Memory Structure

Data from file is directly stored into this circular buffer at appropriate location within the free frame that is at an offset of 32 bytes from starting address of each frame. At the beginning of each frame there is a status field which determines whether the frame is ready to be send or not. Then headers are added to them and the starting address of buffer is passed to NIC for sending. Thus Socket buffer and NIC buffer copies are not generated and memory is saved.

At the receiver end, we use a size configurable circular buffer mapped in user space. This way reading packets just needs to wait for them, most of the time there is no need to issue a single system call. By using a shared buffer between the kernel and the user also has the benefit of minimizing packet copies.

B. Reduced Header Size:

The packet part of the frame contains the entire packet structure including the packet header. Appropriated header can be attached to each packet that is sufficient to send data to other end. For Ex: IP address field can be removed as they are not required in closed network where nodes can be identified using MAC address itself. The header should contain the destination MAC address to identify the destined node. Thus a large part of header processing can be eliminated which can save a lot of CPU cycles

C. Transmission:

When sending starts, NIC starts sending frames in the buffer to the destined node. The sending and filling of frames can be threaded to execute simultaneously. When the buffer gets full, the fill thread is suspended and is resumed when frames get empty. A single send command is used to send a large number of packets present in the buffer. Thus System

calls required are less. When the receiver receives a packet it puts in the buffer and updates the status with a flag. Then the user can read the packet, once the packet is read the user must zero the status field, so the kernel can use again that frame buffer. The user can use poll to check if new packets are in the ring.

III. IMPLEMENTATION

A. Server Side:

Following steps [9] are taken :

a) *Socket()* : Creation of the transmission socket.
b) *Setsockopt()* : Allocation of the circular buffer.
c) *Bind()* : Bind socket with a network interface.
d) *mmap()* : Mapping of the allocated buffer to the user process.
e) *Send()* : Send all packets that are set as ready in the ring.
f) *Close()* : Destruction of the transmission socket.

B. Receiver Side:

Following steps [9] are taken:

a) Setup

a) Socket() : Creation of the capture socket.
b) Setsockopt() : Allocation of the circular buffer.
c) Mmap() : Mapping of the allocated buffer to the user process.

b) Capture

a) Poll () : To wait for incoming packets.

c) Shutdown

a) Close() : Destruction of the capture socket and deallocation of all associated resources.

Figure 4 shows the overall design of the system

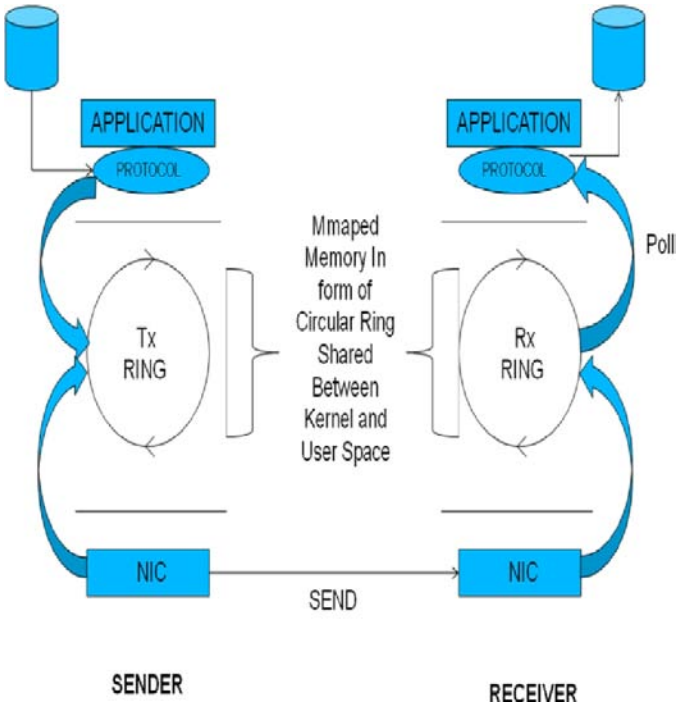

Figure 4. Overall Design

IV. PERFORMANCE

To test, we measured each overhead of the file transmission using QuicX and NFS (TCP was used as default) under various file size and compared with each other. Both the server and client used an Intel Pentium Dual-core CPU, 2GB DDR2 RAM PC with Ubuntu 10.04 Linux(kernel version is Linux 2.6.31.2). The NIC is Realtek 8169 for Gigabit LAN.

Figure 5 shows the throughput of data blocks with size of 1K, 2K, 4K, 8K ... 1M in the test partition disks of server with random reading and writing. Figure 6 shows the server-side CPU load. As comparison, the TCP is included in the two figures. As can be seen from Figure 5, when the data block is relatively small in a single reading and writing, QuicX is worse than TCP. With the increase in onetime reading and write data blocks size, QuicX performance is improving.

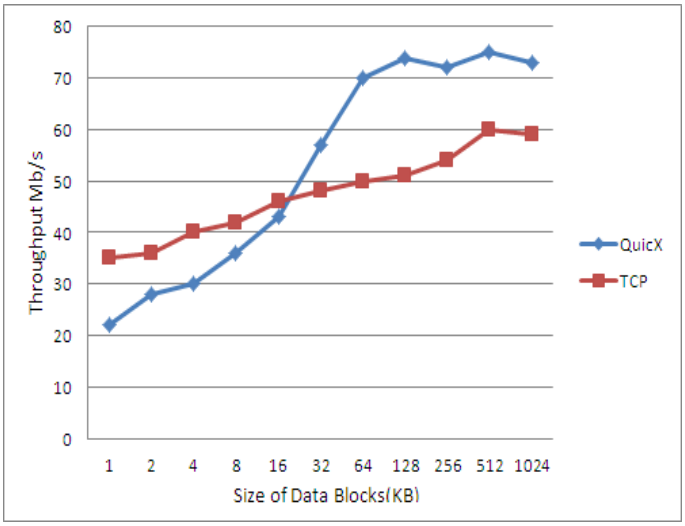

Figure 5. Comparison of throughput under TCP and QuicX

Figure 6 shows the load conditions of server CPU with the trend of the same throughput in Figure 5. With the single-block reading and writing data increasing, the throughput is increasing consequently. In terms of CPU utilization, QuicX is much efficient than TCP.

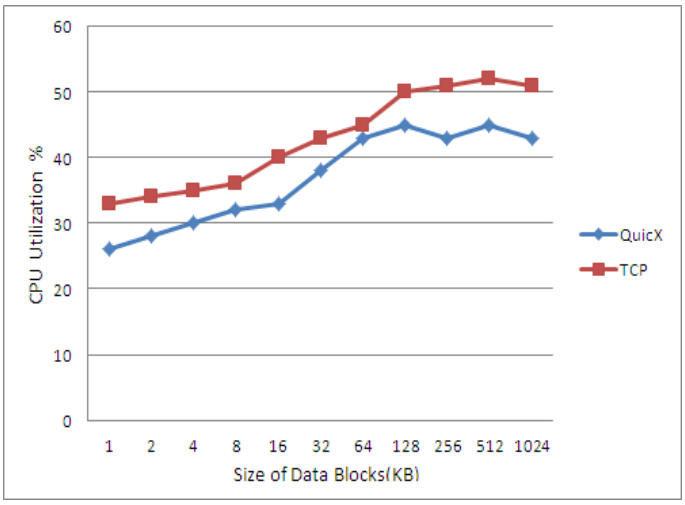

Figure 6. Comparison of server-side CPU load

V. CONCLUSION

To resolve the TCP / IP processing overhead [1] leading to the low performance, the paper design and implement a QuicX Protocol. QuicX drawing on the idea of zero-copy [3][7][11] increases the throughput of the system and reduces the system workload by reducing protocol layers and data copy times. The test result shows that QuicX has better performance than TCP/IP under LAN storage environment. As a light-weight protocol, QuicX can improve the throughout and reduce the system load. QuicX is based on an efficient operating system support for communications in order to provide a cluster of PCs with the capability of efficient parallel processing. It saves valuable system memory, increases bandwidth and decreases latency and ultimately increases system performance.

VI. REFERENCES

[1] D.D. Clark, H. Salwen, V. Jacaobson, J. Romkey, "An analysis of TCP processing overhead", Communications Magazine, IEEE, vol. 20, Issue. 5, pp. 94-101, May 2002.

[2] Antonio F. Díaz, Jesús Ferreira, Julio Ortega, Antonio Cañas, Alberto Prieto , "CLIC: Fast communication on linux clusters", Proceedings of the IEEE International Conference on Cluster Computing, pp. 365, 2000.

[3] Liu Tianhua, Zhu Hongfeng, Chang Guiran, Zhou Chuansheng , "The design and implementation of zero-copy for linux", Eighth International Conference on Intelligent Systems Design and Applications, vol. 1, pp. 121-126, Nov 2008.

[4] Giuseppe Ciaccio, "Optimal communication performance on fast ethernet with GAMMA", Proceedings Workshop PC-NOW, IPPS/SPDP'98, Orlando, FL, Springer, pp. 534-548, April 1998..

[5] M. Welsh, A. Basu,"Low-latency communication over fast ethernet". Proc. Euro-Par'96, Springer, pp. 187-194, Aug 1996.

[6] P.W. Frey, G. Alonso, "Minimizing the hidden cost of RDMA", ICDCS, 29th IEEE International Conference on Distributed Computing Systems, pp. 553-560, June 2009, ISSN:1063-6927, doi: 10.1109/ICDCS.2009.32.

[7] Jerry Chu, "Zero-Copy TCP in Solaris", Proceedings of the USENIX Annual Technical Conference, pp. 253-264, 1996.

[8] Andreas Schaufler, "Linux Network Performance : RAW ethernet vs. UDP", [Online]. Available: http://aschauf.landshut.org/fh/linux/udp_vs_raw/index.html

[9] Sockets: http://www.linuxhowtos.org/C_C++/socket.htm

[10] P. Geoffray, "A critique of RDMA." [Online]. Available: http://www.hpcwire.com/features/17886984.html

[11] Mei-Ling Chiang and Yun-Chen Li, "LyraNET: A zero-copy TCP/IP protocol stack for embedded systems", Journal of Real-Time Systems, Springer Netherlands, Volume 34, Number 1, pp. 5-18, September, 2006

[12] A. F. Díaz, J. Ortega, A. Cañas, F. J. Fernández, M. Anguita, A. Prieto, "The lightweight protocol CLIC on gigabit ethernet", IPDPS '03 Proceedings of the 17th International Symposium on Parallel and Distributed Processing, IEEE Computer Society Washington, DC, USA, pp. 200.1, April 2003.

.