Normalized Transfer of Bulk Data By Using UDP In Dedicated Networks

Kurmachalam Ajay Kumar[1], Saritha Vemuri[2] & Ralla Suresh[3]
[1]Sree Kavitha Engineering College, Khammam, AP., [2]Pulipati Prasad Engineering College,Khammam, A.P &
[3]Pulipati Prasad Engineering College,Khammam, A.P.
Email: ajaykumarcvr502@gmail.com, sarithak.vemuri@gmail.com, rella.suresh@gmail.com

Abstract - High speed bulk data transfer is an important part of many data-intensive scientific applications. TCP fails for the transfer of large amounts of data over long distance across high-speed dedicated network links. Due to system hardware is incapable of saturating the bandwidths supported by the network and rise buffer overflow and packet-loss in the system. To overcome this there is a necessity to build a Performance Adaptive-UDP (PA-UDP) protocol for dynamically maximizing the implementation under different systems. A mathematical model and algorithms are used for effective buffer and CPU management. Performance Adaptive-UDP is a supreme protocol than other protocols by maintaining memory processing, packet-loss processing and CPU utilization. Based on this protocol bulk data transfer is processed with high speed over the dedicated network links.

Keywords- Flow control, high-speed protocol, reliable UDP, bulk transfer.

I. INTRODUCTION

Transmission Control Protocol (TCP) has been used effectively for decades but in the later periods TCP has some shortcomings over wide area high-speed networks. The first shortcoming is Additive Increase Multiplicative Decrease Algorithm (AIMD) congestion control algorithm is poor in discovering available bandwidth and recovering packet loss from high bandwidth-delay product networks. In this AIMD algorithm, bandwidths are distributed equally among the current participants in network and use a congestion control mechanism based on packet loss. Throughput is halved in the presence of detected packet loss and only additively increased during subsequence loss- free transfer.

The second shortcoming of TCP is its congestion window. To ensure sequential delivery both receiver and sender maintains a congestion window for complete size of the buffer. Sender sends a burst of packets and then receiver send back positive acknowledgements in order to receive the next window. Using the throughput time and logic, the sender decides the packets loss in the window and retransmits them to the receiver.

On the contrary network with high-latencies, reliance on synchronous communication can severely arrest any attempt for high-bandwidth utilization because the protocol relies on latency-bound communication.

II. HIGH SPEED RELIABLE UDP

High-speed Reliable UDP Protocols used for bulk data transfer and TCP is used to transfer the control data. Most high-speed reliable UDP protocols use delay-based rate control to remove the need for congestion windows. This control scheme allows a host to statically set the rate and open the throughput-limiting stair step effects of AIMD. In addition, consistent delivery is ensured with delayed, selective or negative acknowledgements of packets. Negative acknowledgements are optimal in cases where packet loss is minimal. If there is little loss, acknowledging only lost packets will gain least amount of synchronous communication between the hosts. Let us consider one high-speed reliable UDP is RBUDP. Describes an aggressive bulk data transfer scheme, called Reliable Blast UDP (RBUDP), intended for extremely high bandwidth, dedicated- or Quality-of-Service- enabled networks, such as visually switched networks.

A Reliable blast UDP

Reliable Blast UDP has two goals. The first is to keep the network pipe as full as possible during bulk data transfer. The second goal is to avoid TCP's per-packet

interaction so that acknowledgments are not sent per window of transmitted data, but aggregated and delivered at the end of a transmission phase. The RBUDP data delivery scheme illustrates the first data transmission phase (A to B in the figure); RBUDP sends the entire payload at a user-specified sending rate using UDP datagrams. Since UDP is an unreliable protocol, some datagrams may become lost due to congestion or inability of the receiving host from reading the packets rapidly enough. The receiver therefore must keep a tally of the packets that are received in order to determine which packets must be retransmitted. At the end of the bulk data transmission phase, the sender sends a DONE signal via TCP (C in the figure) so that the receiver knows that no more UDP packets will arrive.

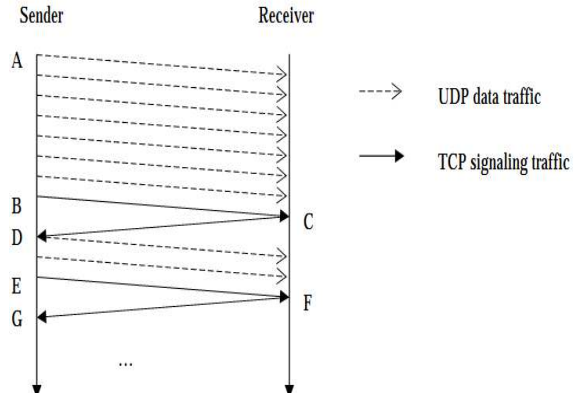


Figure 2.1 The Time Sequence Diagram of RBUDP

The receiver responds by sending an Acknowledgment consisting of a bitmap tally of the received packets (D in the figure). The sender responds by resending the missing packets, and the process repeats itself until no more packets need to be retransmitted. In RBUDP, the most important input parameter is the sending rate of the UDP blasts. To minimize loss, the sending rate should not be larger than the bandwidth of the bottleneck link (typically a router). Tools such as Iperf [Iperf] and netperf [Netperf] are typically used to measure the bottleneck bandwidth. In theory if one could send data just below this rate, data loss should be near zero. The chief problem with using Iperf as a measure of possible throughput over a link is that it does not take into account the fact that in a real application, data is not simply streamed to a receiver and discarded. It has to be moved into main memory for the application to use.

Three versions of RBUDP were developed:

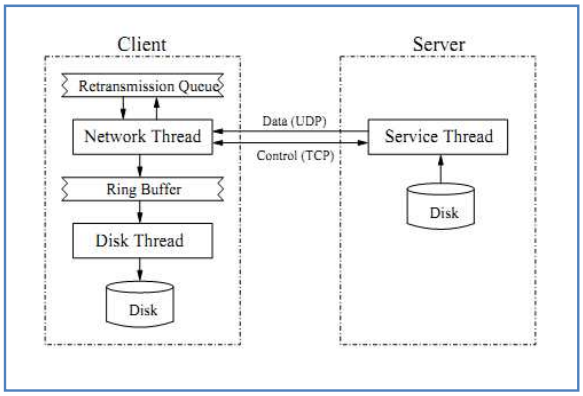
1. RBUDP without scatter/gather optimization – this is a naïve implementation of RBUDP where each incoming packet is examined (to determine where it should go in the application's memory buffer) and then moved there.

2. RBUDP with scatter/gather optimization – this implementation takes advantage of the fact that most incoming packets are likely to arrive in order, and if transmission rates are below the maximum throughput of the network, packets are unlikely to be lost. The algorithm works by using readv() to directly move the data from kernel memory to its predicted location in the application's memory. After performing this readv () the packet header is examined to determine if it was placed in the correct location. If it was not (either because it was an out-of-order packet, or an intermediate packet was lost), then the packet is moved to the correct location in the user's memory buffer.

3. "Fake" RBUDP – this implementation is the same as the scheme without the scatter/gather optimization except the incoming data is never moved to application memory. This was used to examine the overhead of the RBUDP protocol compared to raw transmission of UDP packets via Iperf.

A. Tsunami Protocol

Our initial implementation of the Tsunami protocol consists of two user-space applications, a client and a server. The structure of these applications are illustrated as, during a file transfer, the client has two running threads.



The network thread handles all network communication, maintains the retransmission queue, and places blocks that are ready for disk storage into a ring buffer. The disk thread simply moves blocks from the ring buffer to the destination file on disk. The server creates a single thread in response to each client connection that handles all disk and network activity.

The client initiates a Tsunami session by connecting to the TCP port of the server. Upon connection, the server sends a small block of random data to the client. The client then xor's this random data with a shared secret, calculates an MD5 checksum, and transmits the result to the server. The server performs the same operation on the random data and verifies that the results are identical thus establish client authentication. After exchanging protocol revision codes, the client sends the name of the requested file to the server.

If the server indicates that the file is available, the client sends its desired block size, target transfer rate, error threshold, and inters- packet delay scaling factors. The server responds with the length of the file, the agreed-upon block size, the number of block, and a timestamp. The client then creates a UDP port and transmits the port number. At this point, transmit of the file from server to client is ready.

Finally, reliable UDP is positioned at the application level; this allows users to explore more customized approaches to suit the type of transfer, whether it is disk-to-disk, memory- to-disk, or any other dedicated combination.

III. MATHEMATICAL MODEL FOR MEASURING HIGH- SPEED NETWORKS

A mathematical model is used to determine the changes in system parameters in High-Speed performances. This will be performed by buffer sizes to network rates and sending rates to interpacket delay times. Receiver is very much significant because the receiver is noticed to be in more system strain than the sender. Two buffers are of primary importance in preventing packet loss at the receiving end: the kernel's UDP buffer and the user buffer at the application level.

A. Receiving Application Buffers

For the protocols which receive packets and write to disk asynchronously, the time before the receiver has a full application buffer can be calculated with a simple formula Let "**t**" be the time in seconds, **r (.)** be a function which returns the data rate in bits per second (bps) of its argument, and m be the buffer size in bits. The time before m is full is given by

$$t = m / r(recv) - r(disk)$$

To circumvent this problem, one may put a restriction on the size of the file sent by relating file size to **r (recv)*t**. Let "**f**" be the size of a file and "$\mathbf{f_{max}}$" be its maximum size:

$$f_{max} = m / 1 - r(disk)/r(recv)$$

Note that f_{max} can never be negative since **r (disk)** can only be as fast as **r (recv)**. Also, note that if the two rates are equally matched, $\mathbf{f_{max}}$ will be infinite since the application buffer will never overflow.

B. Receiving Kernel Buffers

Another source of packet loss occurs when the kernel's receiving buffer fills up. Since UDP was not designed for anything approximating reliable bulk transfer, the default buffer size for UDP on most operating systems is very small.

$$t = m/ r(recv)$$

Let **t%** represent the percentage of time during execution that the application is actively receiving packets, and **r(CPU)** be the rate at which the CPU can process packets:

$$t\% >= r(recv)/r(CPU)$$

IV. IMPLEMENTATION OF PA-UDP

PA-UDP is the protocol mostly used in Linux and UNIX environments, this is a multithreaded designed to be self- configuring with minimal human input.

A. Data Flow and Structures

The implementation of the both sender and receiver is accessed as; the sender sends data asynchronously through UDP socket. Periodically probing the TCP socket is used for control and retransmission requests. A buffer is maintained to both sender and receiver. So, the sender does not have to reread the retransmitted packet from disk.

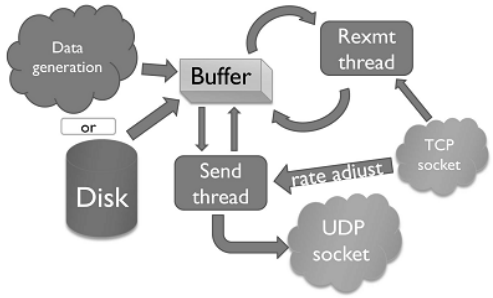


Figure 4.1 PA-UDP: the data sender

When the data are generated, a buffer might be crucial to the integrity of the received data if data are taken from sensors or other such no reproducible events. At the receiver end, there are six threads. Threads serve to provide easily achievable parallelism, crucially hiding latencies (measure of time delay in system). Furthermore, the use of threading to achieve periodicity of independent functions simplifies the system code. The File processing thread ensures that the data are in

the correct order once the transfer is over.

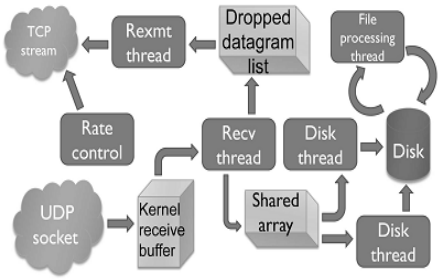


Figure 4.2 PA-UDP: the data receiver

The Recv thread is very sensitive to CPU scheduling latency, and thus, should be given high scheduling priority to prevent packet loss from kernel buffer overflows. The PA-UDP protocol handles only a single client at a time, putting the others in a wait queue. Thus, the threads are not shared among multiple connections. Since, the maximum goal is to utilization of link over a private network.

C. Disk Activity

The disk write thread is very important from a performance standpoint. Writing is done synchronously with the kernel. File streams are default to being buffered, but in our case, this can have unpleasant effects on CPU latencies. Normally, the kernel allocates as much space as necessary in unused RAM to allow for fast returns on disk writing operations. The RAM buffer is then asynchronously written to disk, depending on algorithms used, write-through, or write- back.

No problem to take care if a system call is to write to disk halts thread activity, because disk activity is decoupled from data reception and halting will not affect the rate at which packets are received. Thus, it is not relevant that a buffer be kept in unused RAM. If the transfer is large enough, eventually, this will cause a premature coloring of the kernel's disk buffer, which can introduce unacceptably high latencies across all threads. We found this to be the cause of many dropped packets even for file transfers having sizes less than the application buffers. Two parallel threads are used to write to disk. Since part of the disk thread's job is to hold data together and do memory management, better efficiency can be achieved by having one thread do memory management, while the other is blocked by the hard disk and vice versa. A single- threaded solution would introduce a delay during memory management. Parallel disk threads remove this delay because execution is effectively pipelined. So, by adding of second thread appreciably augmented disk performance is reliable.

Since data may be written out of order due to packet loss, it is necessary to have a reordering algorithm works on putting the file in its proper order.

CONCLUSION

UDP-based protocols are often used in data intensive applications for bulk data transfer. However, it is difficult to implement a new protocol in a fast and efficient way. By introducing a performance adaptive-UDP high speed bulk data transfer is processed between the dedicated network links and this is observed by the mathematical measurements on implementing them on Linux. So, the PA- UDP is consistent than other protocols in CPU utilization efficiency.

REFERENCES

[1] E. He, J. Leigh, O.T. Yu, and T.A. DeFanti, "Reliable Blast UDP: Predictable High Performance Bulk Data Transfer," Proc. IEEE Int'l Conf. Cluster Computing, pp. 317-324, http://csdl.computer.org/, 2002.

[2] A.C. Heursch and H. Rzehak, "Rapid Reaction Linux: Linux with Low Latency and High Timing Accuracy," Proc. Fifth Ann. Linux Showcase & Conf. (ALS '01), p. 4, 2001.

[3] Y. Gu and R. Grossman, "Using UDP for Reliable Data Transfer over High Bandwidth-Delay Product Networks," 2000.

[4] N.S.V. Rao, Q. Wu, S.M. Carter, and W.R. Wing, "High-Speed Dedicated Channels and Experimental Results with Hurricane Protocol," Annals of Telecomm., vol. 61, nos. 1/2, pp. 21-45, 2006.

[5] J. Semke, J. Madhavi, and M. Mathis, "Automatic TCP Buffer Tuning," Proc. ACM SIGCOMM '98, Aug. 1998.

[6] S. Shalunov and B. Teitelbaum, "A Weekly Version of the Bulk TCP Use and Performance on Internet2," Internet2Netflow: Weekly Reports, 2004.

[7] R. Stewart and Q. Xie, Stream Control Transmission Protocol, IETF RFC 2960, www.ietf.org/rfc/rfc2960.txt, Oct. 2000.

[8] S. Floyd, "High-speed TCP for Large Congestion Windows,"citeseer.ist.psu.edu/article/floyd02highspeed.html, 2002.