Experiences in Design and Implementation of a High Performance Transfer Protocol[1]

Yunhong Gu ygu@cs.uic.edu
Marco Mazzucco marco@dmg.org
Xinwei Hong xinwei@lac.uic.edu
Robert Grossman[2] grossman@uic.edu

Laboratory for Advanced Computing, University of Illinois at Chicago
851 South Morgan Street, MC 249, Chicago, IL 60607

Abstract

Applications that transfer huge amounts of data over gigabit networks, such as distributed data mining, have exposed the shortcomings of TCP and have encouraged new protocol's emergence. Common bottlenecks of transfer protocols over gigabit networks include the bandwidth utilization efficiency and the end-system processors ability for processing the data. This paper presents several techniques to overcome these challenges, including new rate control method, avoidance of data copy, and parallel protocol processing. A practical protocol, SABUL, has been designed and implemented based on these techniques. The paper will also address some important issues in implementation that will affect the performance.

Keywords: High Performance Protocol, SABUL, Congestion Control, Parallel Protocol Processing

1. Introduction

High-speed network bandwidth has increased sharply over the past several years. The 1 GigE link has been a popular backbone for many LANs or WANs and 10 GigE is becoming popular. The high-speed network has encouraged a series of new, powerful applications that were impossible in the past, such as remote scientific 3D visualization, immersive interaction, distributed data mining, etc.

A typical use scenario in high performance communication networks is that of a small number of sources sharing the fiber link with huge amounts of data flow. This is quite different than the situation on the public Internet, where there are lots of connections packed in a relatively narrow bandwidth with small data flow for each connection. The difference between the high-speed links and the Internet makes many protocols that work well on the Internet not suitable for the optical link. For instance, the de facto standard transfer protocol on the Internet, TCP, cannot utilize the bandwidth efficiently because of its window based flow control algorithm limit.

The characteristics of high-speed environments require that the protocols should have an effective and fair rate control mechanism, and consume as little processor time as possible. The three research problems addressed by our protocol are listed below.

What is a rate control algorithm that can utilize all available bandwidth while sharing the resources fairly with coexisting connections? The TCP window based flow control dose not work well on optical fiber because the source has sent several gigabits of data into the link before it receives the first acknowledgment that shows packets loss, and following acknowledgments will decrease the window size dramatically.

Researchers have proposed the rate based congestion control [6]. It has several advantages over window based control. First, rate based control can regulate the network flow into a good shape and avoid the burst. Second, rate based control is easier to implement and needs less processor time than window based control. Finally, the new rate control algorithm is expected to utilize the bandwidth more efficiently.

The key problem in rate based congestion control is how to control the sending rate. Sending rate should increase to fulfill the bandwidth when there is no loss and decrease to the same level with coexisting connections and control the loss rate below a limit when loss occurred.

[1] This work is supported by grants from NSF and DOE.
[2] Robert Grossman is also with Magnify, Inc.

How to transparently reduce the memory copy between application and protocol buffer, between user space and kernel space, and between disk and physical memory? Memory copy is a major bottleneck in high performance protocols, which has been recognized by people with experience in the field of web servers [8]. It consumes most of the processor time during the transmission, and when all the processor time is used up the sending rate will be prevented from increasing. In addition, to copy a large data block may cause a receiver process to be suspended and lead to loss packets.

A high performance transfer protocol generally owns a very large protocol buffer for the purpose of data reordering and error correction. Memory copy between this buffer and application memory seriously affect the transfer speed. In a regular use scenario, an application calls a protocol API to receive a large block of data. This will cause a large data block to be copied, which will cost much processor time and even cause packet drops because the processor is too busy to receive the following packets in the data copy stage.

Moreover, most of the current operating systems need to copy the data between user space and kernel space when network API sends/receives data for safety purposes. The overhead of this copy is also serious when the transfer speed is gigabits per second.

Memory copy also occurs in the file transfer, which is not rare over high performance networks. Applications like Grid-FTP, data mining, and scientific visualization are all involved with direct access to data file in disk. This process generally needs to copy data between disk and user space memory.

A high performance transfer protocol needs to avoid the memory copy transparently. By "transparently" it means that the new protocol should keep the same API as the previous version so that applications need not be modified.

How the protocol can transparently utilize multiple processors and multiple NICs at the end of extremely high networks? As the bandwidth increase is faster than the processor speed increase, today a single processor may not meet the requirement of data processing speed when transferring data over extremely high-speed networks. For example, one Pentium 4 2.0GHz CPU at the end system may not feed the 10 Gbps data transfer speed. In such situations people try to write parallel program running over multiple processors or multiple machines to utilize the bandwidth effectively. In addition, nowadays the NIC speed has fallen behind to the network speed such as the optical links with lambda switches. For example, the highest speed NIC in market is 1 GigE, while network speed has been increased to 10 GigE. High performance applications often need to utilize multiple NICs to allocation the bandwidth.

The parallel processing should be moved into the protocol. A protocol that can use multiple processors transparently not only makes the application programming easier, but also has an increased possibility to optimize the processing.

We have designed and implemented a rate-adjusting UDP based protocol named SABUL, or simple available bandwidth utilization library, to solve these problems. SABUL is a lightweight, application level high performance transfer protocol. It uses UDP to transfer application the data and TCP to transfer control information. Features of SABUL also include rate based congestion control, transparent avoidance of data copy at the application level, direct file transfer support, transparent protocol processing, etc.

The rest of the paper examines solutions to these problems and important implementation issues of SABUL in detail. We begin by introducing the SABUL protocol architecture in section 2. Section 3 discusses the rate control algorithms. Section 4 presents the method to avoid data copy. Section 5 offers the architecture of parallel protocol processing. In section 6 we present those important issues when implementing a high performance protocol. The paper is concluded in section 7 with a brief look at future work.

2. An Overview of SABUL

SABUL is a lightweight application level high a performance reliable transfer protocol over UDP and TCP. Application data is transferred over a UDP connection and the feedback from the receiver side is transferred over a TCP connection for reliability.

There are four kinds of packets in SABUL: one kind of data packet and three kinds of feedback packets that carry control information – the ACK, ERR and SYN report. The data packet is a sequence number followed by the application data. The acknowledgment (ACK) report consists of a packet type and a sequence numbers that means all packets previous to it have been successfully received. The loss report (ERR) consists of a packet type, the number of the loss, and the list of sequence numbers of the lost packets. Finally, the synchronization report (SYN) is used to report rate control information, which is a packet type followed by the number of received packets since last SYN feedback.

Sequence Number	Data
32-bit	1 – 1468bit

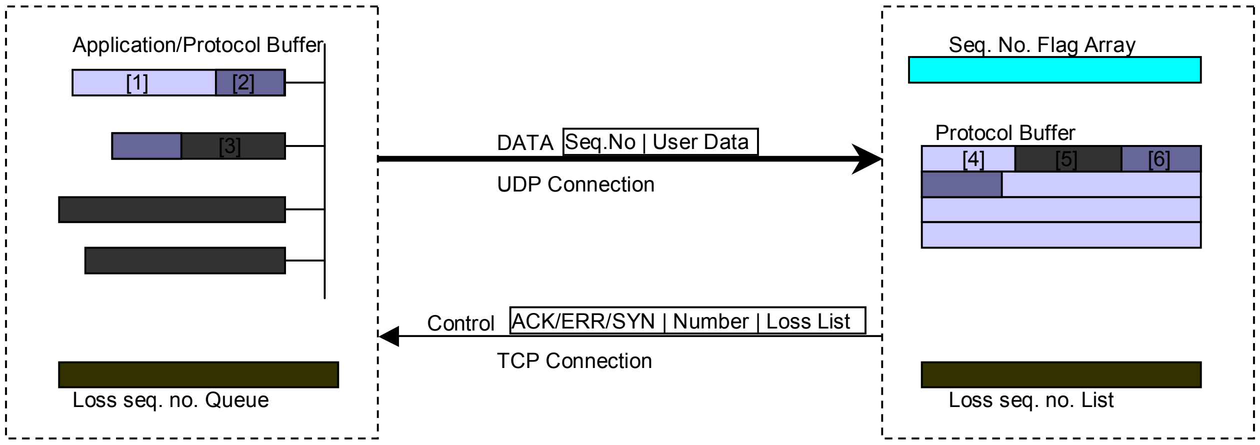
Figure 1. SABUL data packet structure

Packet Type	Attribute	Loss List
32-bit	32-bit	4x ($0 \le x \le 357$)

Figure 2. SABUL control packet structure

In the sender side, application buffers are kept in a list, and loss sequence numbers from the receiver side are kept in a queue. When it is time to send, the sender will try to read a sequence number from the loss queue and send the proper data out; if the queue is empty, a new packet is sent out. When ACK packet comes, the sender moves an acknowledgment flag in the application buffer list. If the flag has reached the end of a buffer, then free the buffer. A timer (EXP timer) is kept in case all packets are lost since the last ACK packet.

In the receiver side, there is a logically circular buffer as the protocol buffer. A flag array is used for loss detection and correction. Arriving packets are written into the buffer and the proper position in the flag array is set. If the new sequence number is greater than the largest sequence number that has been received plus 1, then all sequence numbers between these two numbers will be inserted into the loss list and an ERR report is generated and sent back. ERR report is also activated by a timer in case the retransmitted packets also lost. ACK report is activated either by a timer, by the user registered buffer being full (see section 5), or when the most recent sequence number exceeds the flag array. Once an ACK is generated and sent back, the flag array will shift by a calculated offset and the related pointers in the protocol buffer are set to the new position. The SYN report is activated by timer.



[1] Acknowledged Data; [2] Sent but not acknowledged; [3] not sent; [4] Unwritten area; [5] Written and acknowledged area, data can be read; [6] Written but not acknowledged area.

Figure 3. SABUL Protocol Architecture

The SABUL protocol has been verified and tested. Table 1 lists some benchmarks on both LAN and long delay environments. The detailed performance trace can be found in figure 4.

Table 1. Benchmark of SABUL

Test No.	Band-width	RTT	Sender Configuration		Receiver Configuration		Speed	Loss Rate
			Processor	Memory	Processor	Memory		
1	100M	180 us	Intel PII 500M *2	512M	Intel PII 500M *2	512M	97M	< 0.1%
2	622M	108 ms	Intel Xeon 1.0G *2	512M	Intel Xeon 1.7G *2	512M	580M	< 0.1%
3	1000M	232 us	Intel Xeon 1.7G *2	512M	Intel Xeon 1.7G *2	512M	907M	< 0.1%

We have developed an FTP application (Lambda FTP) that uses SABUL to replace the original TCP data connection. The Lambda FTP is used in optical WANs to transfer large data files like the weather data. Another application is DSTP (data space transfer protocol), which provides a simple means for publishing and querying remote and distributed data. DSTP deploys SABUL as one of its high-speed binary connection types. A streaming data join project has been constructed over DSTP/SABUL, which is used to merge continuous data streams in Gbps rate from distributed places. Results have shown that SABUL works well in these applications.

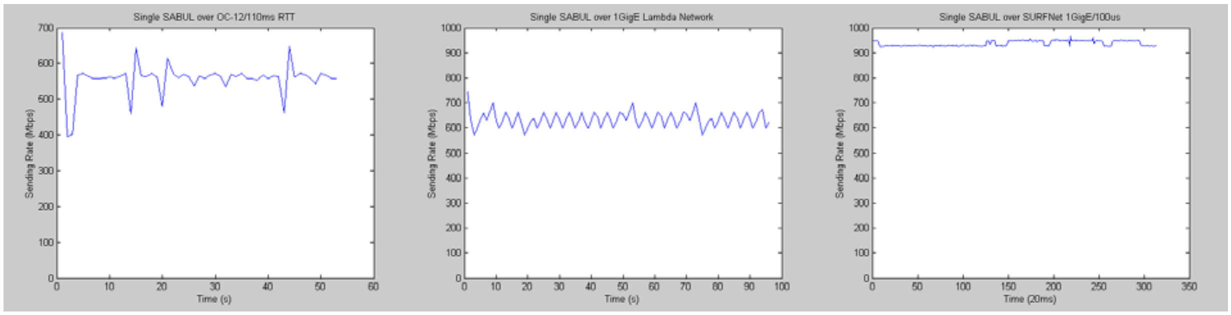


Figure 4. SABUL performance trace

3. Rate Control Algorithm

There are two goals for a rate control algorithm: efficiency and fairness. When the loss rate is below a limit, say 1%, rate-adjusting based protocols should increase the sending rate and eventually reach the bandwidth limit. When the loss rate is above the limit, different rate control algorithms have different efficiency and fairness policies.

First we consider the TCP-friendly fairness. Since TCP is the de facto transfer protocol on the Internet, any new protocols intended to run on the Internet are expected to obey the TCP-friendly rule. TCP-friendly rate control can be reached by simply adjust the sending rate in the same way that TCP adjusts its window size - the AIMD (additive increase, multiple decrease) algorithm: decrease the sending rate to half when loss rate is above the limit and increase a unit when loss rate is below the limit. The rate control interval should be equal to RTT (round trip time). Here we omit the effect of the router, which may drops UDP and TCP packets using different policies [6].

The AIMD algorithm works well when there is no loss – the sending rate will increase until it eventually reach the physical bandwidth limit. However, when congestion occurs, the bandwidth cannot be fully used during the period that sending rate is decreased to half and it is increased to the original value again. The problem is more serious over high-speed links - it takes long time to recover the rate to the optimal value.

In high speed networks there is potential hazard called *loss avalanche*: because too many packets have been sent out when the first acknowledgment indicating loss arrives, those packets that have been sent out will be discarded and generate further loss acknowledgments. This situation causes the sender side to halve its sending rate continuously over several rate control intervals. Figure 3 shows this loss avalanche situation.

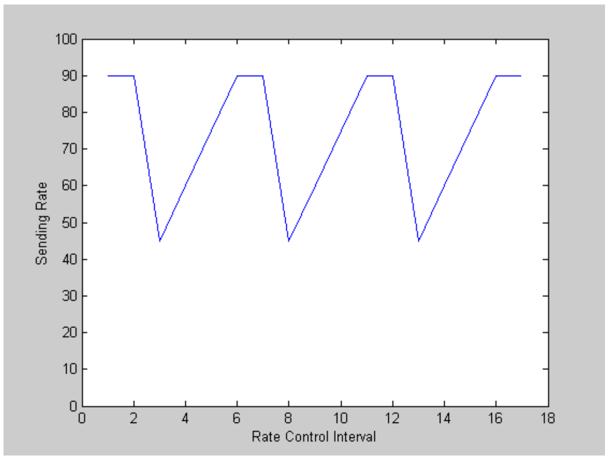


Figure 5. Simulated sending rate change in AIMD algorithm

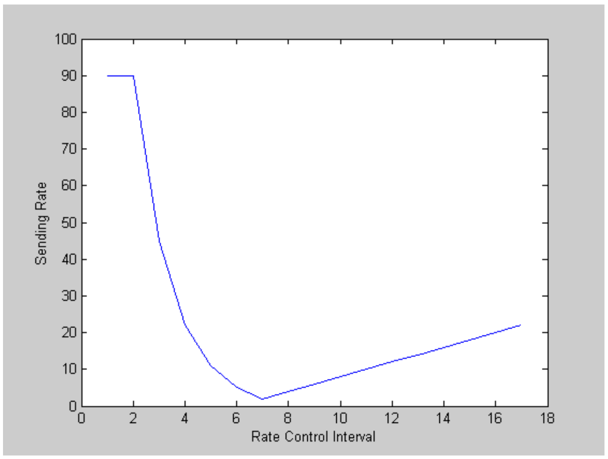


Figure 6. Simulated loss avalanche disaster

Because of these reasons, the AIMD rate control is not suitable for high bandwidth-delay networks.

Second we present a variant of the MIMD (multiple increase, multiple decrease) algorithm. The sending rate is increased or decreased by a factor of the difference of loss rate and loss limit.

$$SendingRate = SendingRate \times (1 + (LossRate - LossLimit))$$

The advantage of MIMD over AIMD algorithm is that in MIMD the sending rate oscillates near the optimal rate, so the average rate is close to the optimal rate.

Another variant of MIMD is to use two limit boundaries, the upper limit and lower limit. The sending rate will not change if the loss rate is between the two boundaries. In an ideal situation the sending rate will remain unchanged until new connections enter or exit such that it can eliminate the oscillations in the above algorithm.

$$SendingRate = \begin{Bmatrix} SendingRate \times (1 + (LossRate - UpperLossLimit)),\ if\ LossRate > UpperLossLimit \\ SendingRate \times (1 + (LossRate - LowerLossLimit)), if\ LossRate < LowerLossLimit \\ SendingRate, otherwise \end{Bmatrix}$$

This algorithm is self-fairness, which means that connections that use same MIMD algorithm will share the link fairly. However, the fairness between this algorithm and TCP is still unexploited[3].

In SABUL we use this MIMD variant as its rate control algorithm with an enhanced mechanism for long delay networks. Since the rate control is based on a timer in constant or variant interval that can be as long as 100ms over long distance links, if congestion occurs there should be some mechanisms to detect it as early as possible. There are two warning events that trigger the early congestion detection in SABUL: 1) continuously receiving ERR reports and 2) unacknowledged packets reached a threshold. In the former case, SABUL increases the sending interval by a constant value, but not more than twice during one rate control interval; and in the latter case, SABUL doubles the sending rate.

Below is the rate control algorithm used in the SABUL protocol.

```
Sender Side:
  on numbers of unacknowledged packet reaching warning threshold
     $r = \min\{t, r + c\}$
  on loss reports (ERR) arriving continuously
     $r = \min\{t, r + c\}$
  on receiving synchronization (SYN) feedback
     $r = \begin{Bmatrix} \min\{t, r \times k \times (1 + (p - \alpha))\},\ if\ p > \alpha \\ \max\{1, r \times (1 + (p - \beta))\},\ if\ p < \beta \\ r, otherwise \end{Bmatrix}$
  where r is the packet sending interval, c is a constant value, p is the current loss rate, α is the upper loss rate limit, β is the lower loss rate limit, k is a constant parameter, and t is the maximum sending interval
Receiver Side:
  feedback current detected packet loss when loss detected (ERR)
  feedback packet loss that the retransmitted packets have not been received at constant interval (ERR)
  feedback total received packet number since last synchronization time at constant interval (SYN)
```

Thirdly we consider the non-linear rate control algorithm. An ideal algorithm should change the rate to the optimal value immediately and keep it unchanged until the loss rate changes. The answer is to find a function that satisfies the relationship of sending rate and loss rate.

[3] Theoretically, MIMD algorithms are more aggressive than AIMD algorithms. However, considering the other factors such as rate control interval, router's treaties to UDP and TCP, etc., the fairness between SABUL and TCP is far more complex. We will do further research on this in the future work.

We tried a curve fitting method to describe the function. Unfortunately, this rate control algorithm is unstable.

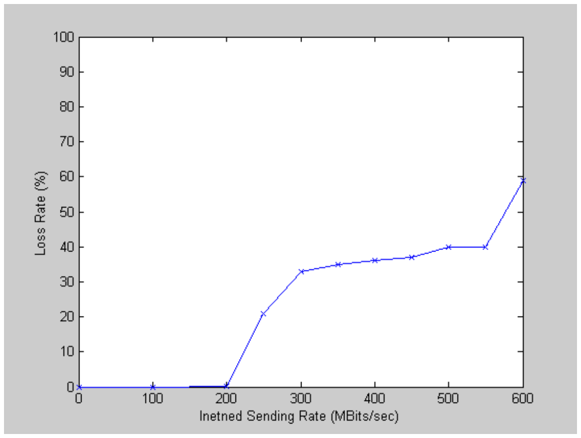


Figure 7. Relationship of loss rate and sending rate

Finally, an important issue is the rate control interval. A short interval leads to a rapid response to the network congestion; however, it also introduces large error in calculating the loss rate and worsens the loss avalanche problem. Traditionally the RTT is used as the rate control interval. However, in high-speed networks the RTT is not large enough especially in LANs or WANs. SABUL chooses a constant value, say 200 milliseconds, as the rate control interval. A variant rate control interval mechanism has been discussed in [10].

4. Avoidance of Memory Copy

Memory copy is one of the major problems in high performance network applications. To send several gigabits of data per second means that at least the same amount to of data should be moved between two memory blocks per second due to the design of current operating system network API. Memory copy consumes much CPU time and prevents the transfer speed to increase when the CPU time is used up.

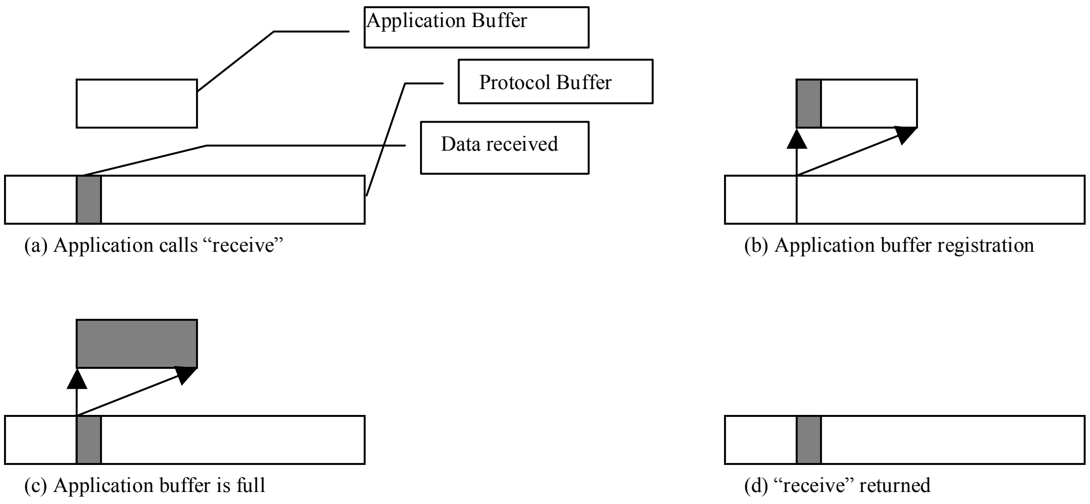
Moreover, an additional buffer is often needed in an application level protocol because the system buffer is too small for applications with gigabits data to transfer. Therefore, there are two kinds of memory copy: the copy between protocol buffer and application buffer and the copy between system buffer and protocol buffer.

To avoid the memory copy we need a method to share the data between different protocol or system levels. In this section we will first discuss the avoidance of memory copy between protocol buffer and application buffer, then list some work done by other researchers at the system level, and finally we will introduce the copy avoidance for file transfer.

On the sender side of the protocol level, the application buffer can be used directly through the locking mechanism, so the avoidance of memory copy is trivial.

On the receiver side if the protocol uses the application buffer directly without changing the API, i.e., transparently, the packets that arrive between two "receive" method calls will be discarded. This is a serious problem. The problem can be avoided with a modification to the protocol API, e.g., ask the application to "provide" some buffer before it calls "receive". It is obvious that the latter solution lacks flexibility. The analysis above shows that the protocol should have its own buffer in the receiver side.

In SABUL we introduced a method called "application buffer registration" as shown in figure 5. This method provides best-effort memory copy avoidance between the application buffer and the protocol buffer. When the application calls the “receive” method to fetch data, SABUL inserts the user buffer into the protocol buffer logically. Before that, all the data from last acknowledgment point to the head point is copied to the user buffer. The insertion of the user buffer is equivalent to extending the protocol buffer by the size of user buffer size. At the same time, the acknowledgment point and the head point of the original buffer are set to the tail buffer, if they are less than the size of the user buffer; otherwise, decrease them from user buffer size, separately. When the user buffer is fully filled, application call returned this buffer.


Application Buffer
Protocol Buffer
Data received
(a) Application calls "receive"
(b) Application buffer registration
(c) Application buffer is full
(d) "receive" returned
Figure 8. Application buffer registration

The memory copy avoidance in this method depends on the applications' behavior. If the application calls "receive" continuously, most of the memory copy can be avoided; otherwise arriving data needs to be written into the protocol buffer first and then copied into the application buffer. It is not a problem for most of the applications to keep the "receive" call continuous as long as the sender side keeps sending. In the SABUL protocol any arriving packet that exceeds the application buffer boundary will activate an ACK feedback, which guarantees that the current "receive" call returns immediately after the user registered buffer is full and reduce the memory copy for the next "receive" call to the least.

In the system level, some researchers have put out several zero copy system level APIs. The emulated data copy [4] uses TCOW (transient output copy-on-write) technology and page swapping to avoid the memory copy between user space and kernel space. Other examples include the fast socket [9] developed in Berkeley. In addition, a well known optimization for memory copy is the data gathering/scattering method, which was been used in SABUL to construct a packet from the data in the different positions in the user space, or to write different packet fields into different user space positions directly.

Memory copy between disk storage and physical memory is also a bottleneck in some high-speed network applications. Many applications read data from files in remote disk and/or store data in local disk, e.g., Grid-FTP [13]. In the regular situation the data needs to be copied from disk into user space memory or from user space memory into disk. This memory copy should be eliminated in the protocol level.

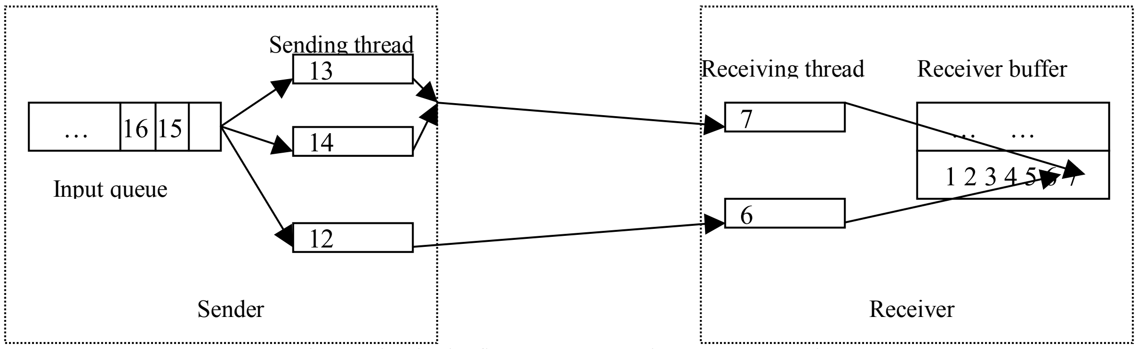
Network researchers and application builders have observed the problem and come up with some solutions. One of the solutions is the introduction of the "sendfile" interface in socket API [8]. Earlier applications before the introduction of this new API often use mapped I/O in application programming for better performance.

We introduced "sendfile" and "recvfile" APIs in SABUL to send and/or receive data from/to disk file directly. In the Linux operating system the implementation uses the "mmap" system call to map a file into virtual memory, and process the data in the virtual memory directly. The performance of this implementation depends on how the system implements "mmap". However, the direct file transfer APIs enables the system and protocol to provide a method to move data directly from hard disk to NIC, or at least to the kernel space memory. Therefore, it avoids the memory copy from disk to physical memory (user space).

5. Parallel Protocol Processing

When processor ability cannot meet the requirements of network transfer speed, a common solution is to use multiple processors in a parallel program. In fact, at the end of extremely high-speed links, there is often a multi-processor system or a computer cluster. Accordingly, multiple NICs are also used to utilize the bandwidth higher than the single NIC speed. Software developers often need to write a parallel program to process the sending and receiving, so it is desirable to implement the parallelism in the protocol level, which can not only provide convenience to the application programming but also can optimize the performance better than at the application level.

The basic idea to use thread level parallelism to distribute the data stream sending/receiving. The input stream (data in the sender side) can be regarded as an FIFO queue. Each sending threads try to read one block from the queue and send out. The reading process is a critical section and each block is assigned a unique sequence number according to the order it is been read. One or more threads can send through one UDP connection and multiple UDP connections can be existed simultaneously according to the number of NICs. At the receiver side, all data block are written into a shared memory between all receiving threads and resembled according to the sequence number. Figure 9 depicts this process.



The number in the figure represents the sequence number

Figure 9: Sending/receiving data in parallel SABUL

The main bottleneck is the reading/writing protection and the synchronization between threads. Since the control information is much less than the user data, we use only one thread to process it. We call the threads dealing with both data and control information the master thread and call the others the slave threads. An application can start exactly one master thread and an optional numbers of slave threads.

In addition, to reduce the overhead of synchronization, in parallel SABUL the packet loss is only processed in master threads. This makes slave threads simple by dealing with new packet sending/receiving only and share as few information as possible.

The number of connections and the number of threads are determined by the network bandwidth, the processors' power, and the NICs. It is naturally to create exactly one connection for one NIC. However, there are situations that sender side and receiver side have different number of NICs. The number of threads depends on the CPU performance and the target transfer speed, but it should not less than the number of connections. These values can be reached using a probing phase, where it starts by only the master thread with one connection, and increases the number of slave threads as well as the connections.

It is obvious that a master-thread-only parallel SABUL is the same as the original SABUL. In fact, the parallel SABUL keeps the same packets specification and processing mechanism as the original one, so it can be used interactively: the parallel SABUL sender/receiver is able to interact with the SABUL receiver/sender, and vice versa.

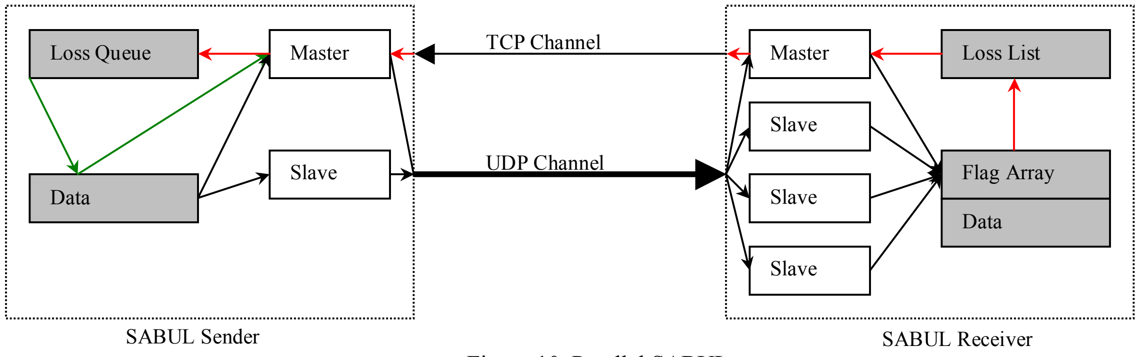


Figure 10. Parallel SABUL

Testing in the environment where the processor ability cannot meet the data transfer speed requirement has shown the parallel SABUL can reach higher performance than the original SABUL.

It should be clarified that the objective of using parallel SABUL is different with those methods of using parallel TCP. The parallel SABUL is to utilize multi-processor and/or multi-NIC transparently, while parallel TCP is to overcome the TCP's low efficiency problem over high bandwidth/delay product networks.

6. Implementation Issues

There are some features not included in the protocol specification but important to the protocol performance in implementation, especially for the high-speed protocols. These issues include timing, synchronization, buffer management, etc. In this section we will examine how SABUL deals with the implementation problems.

In high-speed networks, a source sends out several gigabits per second. Suppose one packet can contain 1500 bytes (MTU of Ethernet), the average packet interval is from several microseconds to tens of microseconds. The problem is in most of the current operating systems there are no such high precision timers. Implementations on such systems have to use busy loop to get the interval.

This busy loop can be avoided when the source can send out data at full speed with a blocked UDP sending. In such a situation a sending call will be blocked until the network is available for the next packet and this blocked time is exactly the expected interval between two packets.

Solutions in the kernel level have been released for high precision timers, including the soft timer [3], the APIC hardware timer on Intel architecture, etc.

Synchronization is also a heavy burden. In SABUL we serialize the operation to the data and the operation to the control information to eliminate the synchronization overhead of the shared data structures between the data processing and the control processing.

The serialization is done by polling the TCP port and setting a timer on UDP receiving. On the sender side, before sending out a packet, SABUL polls the control connection first and process the feedback if there is one; on the receiver side, checks the feedback timers (ACK, ERR, and SYN timers) before receiving packets from data connection, and a timer is set for the receiving in case of a deadlock or delay of the feedback. Overhead of polling a TCP port is much less than the overhead of a mutual exclusion lock (mutex) and the content switch between threads.

In addition, the operation to the protocol buffer, e.g., reading or writing user data, should also avoid locking, because the frequency of this operation may be as high as millions of times per second.

The protocol buffer size also affects the performance. A small buffer is easy to overflow, which limit the sending rate, while a large buffer will delay the response of packet loss and increase the processing time for data reordering and error correction.

Buffer size decides the how many packets can be sent before the first packet gets its acknowledgment and how many packets can be waiting to be reordered and acknowledged on the receiver side before the first packet is moved out from the sequence number flag array. It should be large enough such that the sender can send packets out continuously without interruption and the receiver side should not need to generate too many acknowledgments.

If there is any error between two acknowledgments, the packets from the error point need to wait at least until the next acknowledgment time, hence

$$BufferSize \geq 2 \times SendingRate \times (AcknowledgmentInterval + RTT)$$

It appears that the buffer size should be as large as possible. However, this is not true. Too large of a buffer may cause the same potential hazards as we discussed in section 1 - the loss avalanche. Processing the previous lost packet causes more loss due to the processing delay, which will lead to a chain reaction of packet loss. This phenomenon will continue until the sender decreases the sending rate to a very low level and all the lost packets are received by the receiver.

Moreover, the sender buffer should not be larger than receiver buffer. If the first packet in the receiver buffer is lost and the retransmitted packet has not come before the buffer is full, then all the new packets from the sender will be dropped. All these dropped packets need to retransmit later while after the sender gets the next acknowledgment it will continue to send new data, which may be dropped again.

7. Conclusion and Future Work

Design and implementation of an application level high performance transfer protocol has been desirable as the network bandwidth increases to gigabits. In this paper we describe our experiences with SABUL, a protocol with rate based congestion control over UDP and TCP. We have identified three research problems in high-speed transfer protocols, rate control, avoidance of the memory copy, and parallel processing. SABUL gives efficient solutions to these problems and has shown satisfying testing results.

SABUL has deployed an efficient rate control algorithm and we will continue this theoretical research work. We also identify three kinds of memory copy problems in high performance transfer protocols, including the copy between system buffer and protocol buffer, between protocol buffer and application buffer, and between disk file and physical memory. By now the last two memory copy problems have been efficiently solved the in SABUL.

We will further extend the rate control work to research the fairness between SABUL and TCP. Especially, we hope to address how to define fairness and TCP-friendly fairness? Is there a TCP-friendly rate control algorithm independent of RTT? How does the queuing algorithm in the router affect the UDP stream? Can UDP rate control take advantage of router information?

On the other hand, we will port SABUL into the kernel level of the operating system. We will develop a new UDP API with zero memory copy and uses it in SABUL. Also the copy avoidance between disk and memory will be exploited.

In addition, another variant of SABUL, fast SABUL, will be developed. Fast SABUL is to allow limited data loss to obtain faster transfer speed. Many applications can tolerate data loss, e.g., the high resolution streaming video. Fast SABUL can serve these applications better.

References:

[1] F. P. Kelly, A.K. Maulloo and D.K.H. Tan: Rate control in communication networks: shadow prices, proportional fairness and stability, *Journal of the Operational Research Society 49 (1998), 237-252.*

[2] Deepak Bansal, Hari Balakrishnan: Binomial Congestion Control Algorithms, *Proc. IEEE INFOCOM Conf., Anchorage, AK,* April 2001.

[3] Mohit Aron, Peter Druschel: Soft timers: efficient microsecond software timer support for network processing, *ACM Transactions on Computer Systems*, August 2000.

[4] Jose Brustoloni, Peter Steenkiste: Copy Emulation in Checksummed, Multiple-Packet Communication, *IEEE INFOCOM '97*, April 1997, Japan.

[5] Aled Edwards, Steve Muir: Experiences Implementing A High-Performance TCP In User-Space, *Proceedings of ACM SIGCOMM '95*, September 1995, pp. 196 - 205

[6] P. Pieda, N. Seddigh, B. Nandy: The Dynamics of TCP and UDP Interaction in IP-QoS Differentiated Services Networks, *The 3rd Canadian Conference on Broadband Research*, November 1999.

[7] Jean Walrand, Pravin Varaiya: High-Performance Communication Networks, *Morgan Kaufmann*, 2000.

[8] Philippe Joubert, Robert King, Richard Neves, Mark Russinovich, and John Tracey: High-Performance Memory-Based Web Servers: Kernel and User-Space Performance*, USENIX '2001, Boston, Massachusetts*, June 2001.

[9] Steven H. Rodrigues, Thomas E. Anderson, David E. Culler: High-Performance Local Area Communication with Fast Sockets, *USENIX '97, Anaheim, California,* January 6-10, 1997.

[10] Sally Floyd, Mark Handley, Jitendra Padhye, Jorg Winmer: Equation-Based Congestion Control for Unicast Applications, SIGCOMM 2001

[11] Jeonghoon Mo, Richard J. La, Venkat Anantharam, Jean Walrand: Analysis and Comparison of TCP Reno and Vegas, *INFOCOM 1999*

[12] M. Allman, V. Paxson, W. Stevens: TCP Congestion Control, *RFC 2581*, April 1999

[13] B. Allcock, J. Bester, J. Bresnahan, A. Chervenak, I. Foster, C. Kesselman, S. Meder, V. Nefedova, D. Quesnel, S. Tuecke: Secure, Efficient Data Transport and Replica Management for High-Performance Data-Intensive Computing. *IEEE Mass Storage Conference*, 2001.