National Center for Data Mining / Laboratory for Advanced Computing

Technical Report NCDM/LAC-TR-2002-0001

SABUL: Simple Available Bandwidth Utilization Library

May 2002

University of Illinois, Chicago

© 2001, 2002, University of Illinois

STATUS OF THE MEMO

This technical report describes the SABUL protocol designed and developed in the Laboratory for Advanced Computing, University of Illinois at Chicago. There have been 4 versions of SABUL protocol developed by 5 members of LAC in the past 3 semesters, with assistances from many other members. Please refer to Appendix C for more information. Readers outside LAC please read Appendix B for copyright information.

The currently SABUL release version reflected in this document is 2.0.

This technical report is NOT for publication. Contents in this document may change as new modifications are made to the SABUL protocol; hence, THIS DOCUMENT SHOULD NOT BE CITED OR QUOTED IN ANY FORMAL DOCUMENT.

ABSTRACT

This technical report describes SABUL, the Simple Available Bandwidth Utilization Library, an application level transport protocol for transporting huge amounts of data over high-speed networks. SABUL can utilize bandwidth efficiently while keeping the same API semantics with popular socket interfaces. A variation of SABUL, named parallel SABUL, can also utilize multiple processors to meet the computation requirement over extremely high bandwidths. Generally, SABUL uses UDP to send user data with rate control, and TCP to send control information; it is also designed to avoid memory copy and provide multi-processor support to increase the performance.

TABLE OF CONTENTS

1. Introduction

Simple Available Bandwidth Utilization Library (SABUL) is an application level protocol for emerging high-speed links, like optical fiber with 10G bandwidth. While traditional TCP is not suitable for such networks, SABUL is designed to utilize bandwidth efficiently with high performance computing applications, including grid computation, distributed data mining, etc.

SABUL uses User Datagram Protocol (UDP) as its data channel. User data are packed and sent out at a rate changed according to the network situation (e.g. loss rate). Control information, including acknowledgment and error report is sent through a channel using Transfer Control Protocol (TCP). Rate control is used here to act as a replacement to TCP window control for the purpose of congestion control; however, it shows better bandwidth utilization than window control.

Avoidance of memory copy is another principle in the design of SABUL. On both sides of the transmission SABUL has memory management to process user data so that memory copy can be avoided as much as possible. At the same time, SABUL keeps the same API semantics as traditional socket interfaces.

To deal with the case in which the processor cannot meet the process ability required by the data rate, Parallel SABUL (section 6) can utilize multiple processors to reach higher bandwidth utilization.

Although SABUL is designed for high-speed networks and shows better performance than TCP, it works well over low bandwidth. In addition, SABUL is not only for large block data transmission. Data in small pieces can also be transported with SABUL, with a performance similar to that of TCP.

SABUL is a pure application level protocol, so its specification does not include anything in the lower level protocol and operating system, especially the avoidance of memory copy in the kernel level. The latter is a critical technology in high performance systems, but it is beyond the scope of SABUL. SABUL works on IP networks and it is independent of the implementation of the systems it works on.

Several applications using SABUL have been developed to examine the capability of SABUL, including SABUL-FTP and DSTP. SABUL-FTP is an implementation of File Transfer Protocol (FTP) but it uses SABUL as its data connection protocol. DSTP (Data Space Transfer Protocol) is a protocol used for distributed data mining, which is running over gigabit networks with huge amount of data to transfer frequently.

The rest of the document goes as follows. Section 2 will examine what kind of applications can utilize SABUL for better performance. Section 3 includes the terminology used in SABUL (and this document). Section 4 gives the specification of packet structures of SABUL. The details of SABUL and Parallel SABUL protocols are described in section 5 and section 6, separately. SABUL provides direct support for file transfer, which is discussed in section 7. Section 8 is a specification of SABUL application programming interfaces. Constants used in SABUL are listed in section 9. Finally, section 10 gives the formal specification of SABUL and its formal verification. Some of the algorithms are listed in Appendix A in C++ format.

2. SABUL Use Scenarios

SABUL can utilize bandwidth more efficiently than TCP especially over links with high bandwidth and long delay. SABUL will typically be used over high-speed networks, LANs, and WANs for the transfer of huge amounts of data, like data from a weather center. Here we present two applications already built: the SABUL-FTP and the DSTP protocol.

It must be pointed out that SABUL is not for bulk data transfer only, although it is inspired from those applications involved in huge amounts of data transportation. SABUL can transfer data in small pieces as well, even for those real time applications without hard time constraint.

2.1 Typical Use of SABUL

SABUL was original designed for bulk data transfer and this feature is still a major requirement during the further modification. SABUL can accept user buffer for sending or receiving as large as its own buffer size. However, any given buffer with size greater than the protocol buffer size will be refused and return error. The size limit can be read or set from SABUL interface get/setOpt().

To gain best performance, receiver side application should give the user buffer to SABUL as early as possible. This is because that SABUL uses best effort memory copy avoidance (see section 5.3.1). If possible, try to use continuous "recv" call in the receiver side and use large receiver buffer.

There is no lower limit for the size of the user buffer. Using small piece of buffer will not affect the transfer speed before the CPU time is used up. However, using small buffer very often instead of using a large buffer will consume more CPU time because of the user command process and less possibility to utilize the user buffer registration method. This limitation exists in almost all network APIs, including the regular socket API.

SABUL will return the received buffer to user as soon as the last packet for the buffer is received.

A variant of SABUL, named fast SABUL, is included in the further development plan. Fast SABUL will permit limited packet loss (say, 1%) to obtain faster performance and make SABUL suitable for some real time application that can tolerate data loss, e.g., high resolution streaming video.

The connection phase of SABUL is long comparing to TCP. Applications should avoid disconnecting and reconnecting a SABUL connection very often.

Multicast and security are not considered by the SABUL protocol and not included in near future plan either.

2.2 Case Study: SABUL-FTP

FTP is one of the most popular protocols for sharing data. Traditional FTP uses TCP as its data transfer protocol. Problems of TCP also prevent FTP from utilizing the bandwidth efficiently.

SABUL-FTP uses two SABUL connections to replace the TCP data connection in original FTP. It constructs the SABUL connection as soon as a user connects to the server, and keeps it until the user quits. This is not similar to the original FTP, which may open and close TCP data connection several times during the FTP session. SABUL-FTP only supports the binary connection type in FTP, which is the most often used connection type in FTP sessions. In addition, the direct file operation support API has been first used in this application – the purpose of this feature of SABUL meets the requirement of FTP exactly.

SABUL-FTP has reached the expected performance during testing over 100M LAN, 1G LAN and OC-12 long distance networks. Considering that today there is lot of data file with several megabytes or gigabytes in size, the SABUL-FTP is expected to improve the usability of the FTP protocol.

The SABUL-FTP is also called the Lambda FTP, for the reason that it is running on lambda switch optical links.

2.3 Case Study: DSTP

Data Space Transfer Protocol, also called DSTP, is a protocol for distributed data mining developed in LAC. DSTP servers maintain a metadata information table for a client to search useful data and retrieve it back to a local machine. Most of the data files in the servers are huge, like the weather data, earth data, etc.

Recently LAC members have inserted the SABUL connection type to DSTP protocol. A typical DSTP session is: the client sends a request to construct a binary connection for data transfer with a type information for the connection type, e.g., SABUL; the server responds to the request and opens a SABUL sender, and sends the SABUL sender port number to the client; the client then opens a SABUL receiver and connects to the sender in server side; during the following life cycle client can send request to the server to retrieve data files and the server sends them through the SABUL connection.

Testing has shown that DSTP performance over a SABUL connection is better than the parallel TCP connection, which is another binary connection type available in DSTP.

In addition, we also built an application of streaming data join over DSTP. In this application two real time data streams are sent from the servers in different places and one client receives them and join (merge) the streams into one.

Since SABUL receiver has its own buffer and it starts a thread for receiving, the stream data join application needs not to care the threading of data receiving. The design and implementation of SABUL enables the application has enough CPU time to process the data merge.

3. Definitions

SABUL	Simple Available Bandwidth Utilization Library
Parallel SABUL	A variant of SABUL that can utilize multiple processors transparently
Fast SABUL	A variant of SABUL that permit limited data loss but can improve the performance by reducing part of the retransmission
Channel	SABUL builds two connections for data and control information, separately. Such a connection is called a Channel. The two channels are Data Channel and Control Channel respectively.
Port	Each channel needs one network port on each side. Ports used by the Data Channel are called Data Port, and those used by the Control Channel are called Control Port. The sender side Control Port is known to the application and the others are transparent.
Sender	SABUL is uni-directional in nature. The side that sends data out is named as Sender.
Receiver	The side that receives data is a Receiver.
Buffer	There are three kinds of buffers in SABUL: the UDP buffer, the SABUL buffer and the application buffer. Buffer size and management of the SABUL buffer are important to the protocol performance. The SABUL buffer is also called the protocol memory.
Data Packet	User data is packed into a packet with a 32-bit sequence number and transmitted over Data Channel. Such packets are Data Packets.
Control Packet	The feedback from the receiver side is transmitted using TCP. These feedback packets are Control Packets in SABUL.

4. Packet Structure

There are two kinds of packets in SABUL: Data and Control Packets. Data Packets is for user data to transfer over Data Channel. Control Packets is the receiver side feedback over Control Channel, including positive feedback (Acknowledgment) and negative feedback (Error Report).

Data Packet is a 32-bit sequence number followed by user data:

Sequence Number	Data
32-bit	1 – 1468bit

Note that there is not a field for data size. This information is recorded in UDP head.

The structure of Control Packet shows as following:

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

The packet type filed can be ACK, ERR or SYN, representing acknowledgment, error report, and rate control information, respectively. For ACK packet, attribute field is the sequence number before which all data packets have been received; loss list field in an ACK packet is 0-bit size. ACK packet means all the packets before, but not including, the sequence number recorded in the attribute field have been received. For ERR packet, the attribute field is the number of sequence numbers of lost packets carried in this packet, while the sequence numbers are recorded in loss list field. Finally, for SYN packet, the attribute field is the number of received packet since last SYN packet, and the loss list filed is 0-bit size.

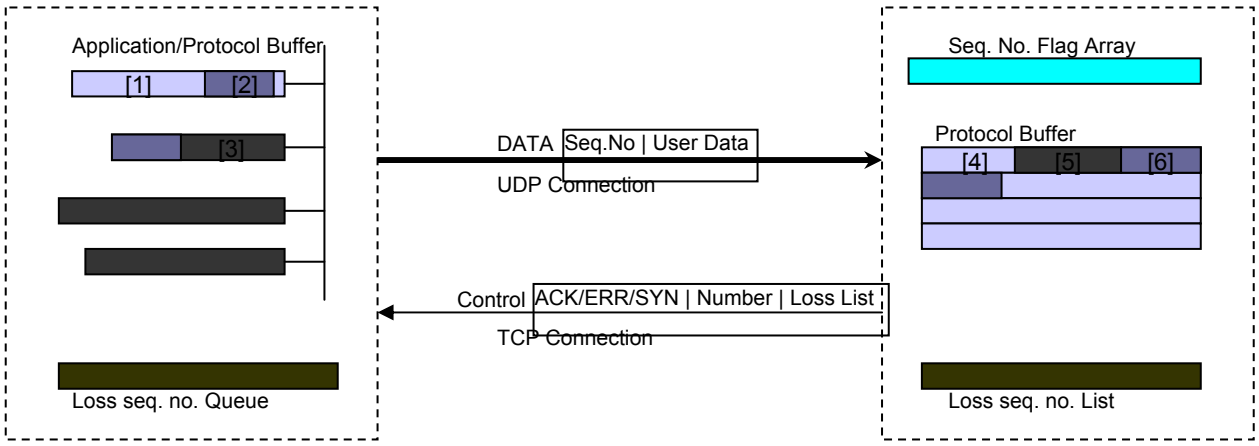
5. Protocol Specification

5.1 General Description

Generally speaking, SABUL is an application level protocol over IP networks. It uses UDP to transfer data with rate control. All the control information needed by error correction, rate control, etc., is transferred over TCP.

The idea above, together with the goal of designing a high performance protocol, brings three design questions to SABUL: how to manage the protocol and application buffer to decrease the overhead of data operation while keeping the traditional API; how to control the data sending rate to reach as high speed as possible while not being too aggressive; how to organize these parts into a protocol without deadlock and synchronization problems.

Each SABUL connection is uni-directional: data can only be sent from sender side to receiver side. The sender side initializes the connection by waiting at a TCP port and after a TCP connection request comes from a receiver; both sides construct a UDP connection as Data Channel. The sender then fetches data from its buffer and sends it out at a rate that may be changed every synchronization interval based on the loss rate. Receiver side maintains an array for re-ordering the coming packets, finding out the next expected sequence number and the lost packets. It also maintains a buffer cooperated with application buffer. ACK and ERR packets are generated in constant interval if there are data received right or there are data lost. Other events can also generate these feedbacks, e.g. when the buffer is full an ACK is generated. SYN packet is generated every constant interval in the receiver side to keep the sender side informed of the receiving rate. In addition, the sender has a timer to monitor the expired event.



[1] Acknowledged data; [2] Sent but unacknowledged data; [3] Unsent data;
[4] Unwritten buffer; [5] Received, written and acknowledged data; [6] Received, written but unacknowledged data.

5.2 Sender Side Protocol

A SABUL sender accepts application data, stripes it into packets and sends the packets to a Receiver. Sender maintains a protocol buffer, a list for lost sequence number, and their management components.

5.2.1 Sender Side Memory Management

The sender side buffer is a list of application data buffers with an upper size limit. Each node of the list (a block of buffer) is the data block that the application calls SABUL to send. The buffer is freed by SABUL after it is been sent successfully.

A SABUL function call to send data can be either blocked or non-blocked. A blocked call adds the buffer to send to the list and waits until it is freed (after sent successfully). A non-blocked call adds the buffer and returns immediately. Both of the calls will return immediately with fail information if the new buffer makes the total buffer size exceed the limit.

There are four pointers in sender's buffer: the current ACK block pointer (CA), the current ACK pointer (CAP), the current sending block pointer (CS), and the current sending pointer (CSP). When CAP is greater than or equal to the size of CA, CA is freed and moved to next block if there is one, and CAP are calculated to represent the position in new CA. CA is the first block in the list, since all the blocks before it have been freed. CA and CAP are also used to find a data position that is to be re-transmitted. CS and CSP are used to find the next data position to send.

SABUL always tries to read data in fixed size and shape fixed-size data packets. However, if the last packet of a buffer is less than the desired size, SABUL sends out a small packet and the receiver will find it out through the packet size.

5.2.2 Data Sending Algorithm

Data Structure:

1. The application/protocol buffer is linked list with its nodes are the head pointer of the application buffer.
2. The loss sequence number queue is a FIFO queue that stores the loss sequence number from ERR feedback.

Algorithms:

STEP 1. Initialization:
STEP 2. Get current time;
STEP 3. Poll Control Channel. Receive control packet if there is one. Receive and process any feedback. Otherwise, calculate the time passed since last ACK or ERR feedback packet was received, if the interval is greater than the time expiration interval, generate an EXP packet and process it.
STEP 4. If the loss queue is not empty, read (and remove) a sequence number and find its position in sender buffer, pack a data packet and send it out. GOTO STEP 6;
STEP 5. Increase the current sequence number; if it is greater than the maximum sequence number, set it to zero. Find next data sending position, if found, pack a data packet and send out;
STEP 6. Get current time and calculate the time passed since STEP 2, if it is less than current sending interval, wait until the time passed to reach the interval. GOTO STEP 2.

Reference:

Appendix A.3

Note that although SABUL does not includes any memory copy avoidance technology in the kernel, it is designed to use application level data scatter/gather technology to avoid the copy overhead of constructing a data packet by moving different parts together.

5.2.3 Feedback Processing

Feedback processing is an independent part in sender, however, it is synchronous, i.e., when feedback processing is called in sender, sender will block until the processing finishes.

There are 4 kinds of feedback: 2 from receiver (ACK and ERR), and another 2 are pseudo feedback generated by sender itself (SYN and EXP).

ACK processing: ACK means all the packets before the sequence number carried by it have been successfully received. ACK processing sends a message to sender buffer telling the latter to modify the acknowledgment pointer and free user buffer if all data in that buffer are acknowledged. ACK processor calculates an estimated acknowledge size by simply multiplying numbers of packets and the fixed packet size. When the buffer processing the acknowledgment, it needs to adjust the size when it meets the end of a buffer and the last packet is less than the desired size.

ERR processing: ERR means some packets were lost. ERR processor inserts the sequence numbers of lost packets to loss list.

SYN processing: SYN processing is for rate control. During the interval between two SYN events, the number of lost packets from ERR and EXP packets are summarized as the total loss number in this interval. The local loss rate can be calculated from the local loss number and local data sending number. The data-sending rate is modified according to the rate control algorithm, which takes the loss rate as input.

EXP processing: EXP means all the packets since the last acknowledgment packet were lost. EXP processor adds these sequence numbers to the loss list.

See Appendix A.2 for reference.

5.2.4 Timing

SABUL dose not have an accurate timer (and it does not need to). All timers are kept by calculating the interval from last event: if the interval is greater or equal to the interval set by SABUL, it means a timer expired event occurred.

The sending rate should be kept firm. In the case that the local host processing ability (processor, memory, bus, and/or disk) is greater than network transmission ability, each sending cycle time is less than the expected sending interval, and SABUL waits the amount of rest time at the end of each cycle. Otherwise, SABUL dose not wait any time and the data transfer rate will be decreased due to the system bottleneck.

On high-speed systems, the interval of two data sending cycle is as small as several microseconds. Most of the current operating system cannot provide such high precision timers. Different implementation may take different methods to solve the question, including blocking sending, busy loop, hardware timer, soft timer, and real system, etc. SABUL protocol does not specify the detail of lower level implementation.

Timing issues in this paragraph are also suitable to SABUL Receiver.

5.2.5 Sequence Number

The 32-bit sequence number is not finite, so special care should be taken when the sequence number grows to the upper bound.

SABUL uses 32 bits for a sequence number and in SABUL the largest sequence number is 2^{31}. When comparing two sequence numbers, a threshold value 2^{30} is used. If the absolute difference between the two numbers is less than the threshold, than the larger number is regarded as the later, otherwise the smaller one is the later one.

Sequence Number issue in this paragraph is also suitable to SABUL Receiver.

5.2.6 Rate Control

The SABUL rate control has three parts that occur on receiving SYN feedback, on sender buffer exceeding warning line and on continuously receiving ERR feedback, separately.

The first one is the regular SABUL rate control that has two phases: the acceleration phase and the regular phase. The acceleration phase starts from the beginning of the connection. During the acceleration phase, the sending rate is doubled or halved to reach the desired rate. During the regular phase SABUL takes an MIMD (multiple increases, multiple decreases) like rate control strategy.

The rate control algorithm takes loss rate since last rate control event and the current sending interval as parameter. And it returns a new sending interval.

STEP 1. Calculate smooth loss rate average (LRA):
LRA = LRA * weight + lossrate * (1 - weight)

STEP 2. If currently it is in acceleration phase: if LRA is greater than the upper loss rate bound, if the new interval is greater than interval limit, set the new interval to the limit, and finish acceleration phase; if LRA is less than the lower loss rate bound, halve the interval and return, if the new interval is less than 1, set the new interval to 1 and finish the acceleration phase; otherwise, return and finish the acceleration phase.

STEP 3.

a. If the LRA is greater than the upper bound, calculate new interval using:
Interval = Interval * (1 + upperbound –LRA)

b. If the LRA is less than the lower bound, calculate new interval using:
Interval = Interval * (1 + lowerbound –LRA)

STEP 4. Cut the interval into [1, limit] and return.

Reference:
Appendix A.1

The other two rate control events are to respond the network congestion as early as possible.

When the sender has continuously received ERR feedback more than a constant warning threshold, the sending interval is increase by a constant value. This step should not be done more than twice between two SYN events:

Interval = Interval + c

When the number of unacknowledged packet has exceeded a warning threshold, the sender should increase the sending rate, by a constant ratio, say 50%. This step cannot be done more than once before the unacknowledged packets number drops below the threshold.

Interval = Interval * 2

This rate control method is not TCP-friendly in its traditional meaning. However, the AIMD based flow control is one of the reasons that cause the low performance of TCP[1].

The traditional TCP-friendly rate control can be easily reached by estimating a TCP link's bandwidth under the same loss rate. Then SABUL can tune its rate to reach same bandwidth utilization. TCP bandwidth utilization estimation can be got through the formula below:

Bandwidth = 1.22 * MTU / (RTT * sqrt(Loss))

5.3 Receiver Side Protocol

A SABUL Receiver receives data packets and re-orders them. When an application calls a receiver's method to receive data, it returns data it has received. Receiver maintains a buffer for temporally storing

[1] In fact, an AIMD rate control in SABUL is not TCP-friendly either. The bandwidth utilization of TCP depends on RTT while SABUL dose not, which is a major difficulty in defining SABUL's fairness. This work is still on going.

data before applications fetch it, a sequence number array for re-ordering and correcting, a list for recording special packets that contains less data than regular packet, and a loss list for data correcting.

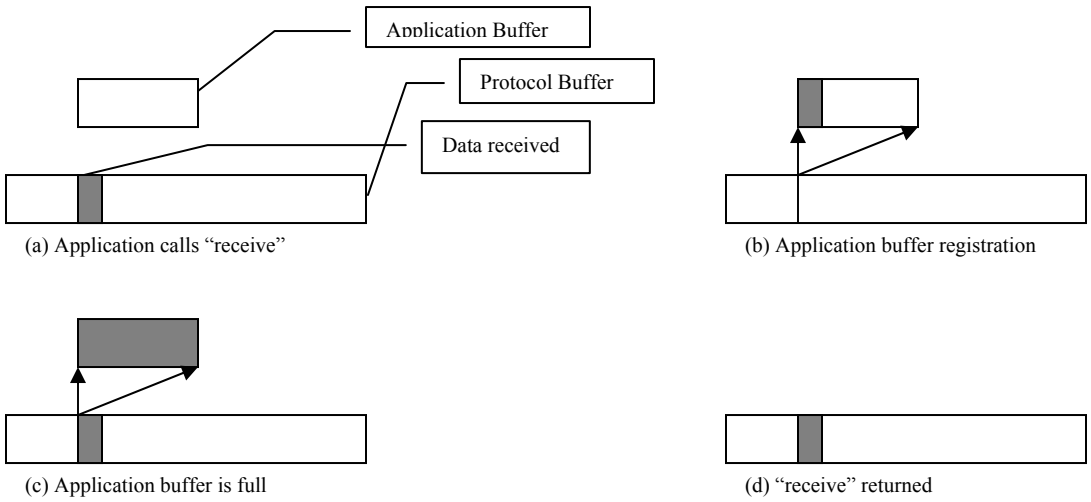
5.3.1 Receiver Side Memory Management

A receiver side buffer is a block of continuous memory, and is logically circular. Two pointers tell the head (the further position of new data, i.e., the data that has the largest sequence number currently) and tail (from which point the application can read data), separately. An additional pointer is used to record the position before all the data was received correctly – the acknowledgment pointer.

Memory management provides functions including: a) positioning a writing slot according to the offset of the current sequence number and the last acknowledged sequence number; b) Moving the buffer for some offset when an out-of-order packet was received and its size is less than the regular size, i.e., the writing slot with a larger sequence number received before the packet has been assigned wrong slot to written. c) Update the current furthest position that has been written, i.e., the head pointer.

One of the goals of the design of the receiver management is to provide transparent memory copy avoidance between protocol memory and application memory. Transparent data copy avoidance means avoiding data copy without change the API semantics. SABUL uses a method of user buffer registration.

When application calls receive method to fetch data, SABUL insert the user buffer to the protocol buffer logically. Before that, all the data from the last acknowledgment point to the head point to be copied to the user buffer. The insertion of the user buffer is equivalent to extend the protocol buffer by the size of the user buffer size. At the same time, the acknowledgment point and head point of the original buffer are set to the tail buffer, if they are less than the size of user buffer; otherwise, decrease them from user buffer size, separately. When the user buffer is fully filled, application call returned with the buffer. See Appendix A.6 for reference.



(a) Application calls "receive"

(b) Application buffer registration

(c) Application buffer is full

(d) "receive" returned

User buffer registration provides the best effort to avoid memory copy. Only the data received between two receive call needs to be copied, so if user calls the receive method continuously, the copy overhead will be very small. It is obvious that if the technology is enabled, only the blocking receiving call can be used.

5.3.2 Data Receiving Algorithm

Data Structure:

1. The protocol buffer is a logically circular buffer for temporally storing and reordering the received packets, which is generally large enough considering the data rate.
2. The sequence number array is a bit array that records the packets with which sequence numbers are received and with which are not.

3. The loss sequence number list is a linked list whose node is a structure of the lost sequence number and the last error report time of the sequence number.

Algorithm:

STEP 1. Initialization:

STEP 2.

a. Get current time;

b. If the time passed since the last acknowledgment event, or the user registered buffer has been fulfilled, generate an ACK packet;

c. If the time passed since the last error feedback event, and the loss list is not empty, generate an ERR packet;

STEP 3. Find a data slot in the protocol buffer for an expected packet; if it failed, use a temperate buffer;

STEP 4. Receive a data packet; if the timer is expired, go to STEP 2;

STEP 5. Read the sequence number in the data packet, and calculate the offset since the last acknowledged sequence number;

STEP 6.

a. If the offset is greater than or equal to the sequence number array size, generate an ACK packet; if the offset is less than 0, go to STEP 2;

b. If the offset is greater than expected offset, insert sequence number from current largest received sequence number to the current received sequence number to the loss list, generate an ERR feedback.

STEP 7. Calculate the next expected offset number. The next expected offset number is the smallest number since the most recent offset (current) in offset flag that not been set;

STEP 8. If the offset is not equal to the expected offset, or the next data slot was not found in STEP 3, rewrite the data to the right position in the buffer; if it failed, go to STEP 2;

STEP 9. Modify the largest received sequence number;

STEP 10. If the size of the current received packet is less than the regular size, insert the sequence number and size into the irregular packet list;

STEP 11. Set the proper slot in offset flag, go to STEP 2.

Reference:

Appendix A.7

In STEP 8 ERR feedback generation can wait to the next loop to avoid out-of-order packets.

5.3.3 Feedback Generation

ACK Generation: If the loss list is empty, the ACK sequence number is the largest received sequence number plus 1; otherwise, the first sequence number in the loss list is the ACK sequence number.

ERR Generation: The principle of ERR generation is: loss information should feedback to the sender as soon as possible; the same loss sequence number should NOT feedback more than once in a RTT time plus the necessary system processing time for a packet. Every node in the loss list is attached with a timestamp. During the ERR generation, only the sequence number on which node the difference of current time and the timestamp is greater than the valve will be added into the ERR packet. Timestamp is updated if the proper sequence number is selected. If the number of loss sequence numbers is greater than the maximum number a packet can carry, stop the procedure.

SYN Generation: SYN feedback is to tell the sender how many packets (including the discarded packets by receiver) have been received since the last SYN feedback. The SYN feedback simply sends back a packet with a SYN packet type followed by the total number of received packets since the last SYN event.

See Appendix A.4 for reference.

6. Parallel SABUL Protocol

Parallel SABUL is useful when single processor ability does not meet the requirement of the data-sending rate (according to network bandwidth). Parallel SABUL is designed to use multiple processors on both sides to increase the data processing ability.

The design of parallel SABUL is based on the following 3 observations: a) Only 1 data connection and 1 control connection are necessary, and SABUL can start multiple threads dealing these connections; b) Sender side and receiver side can have different number of threads; c) One thread is enough to process control information.

In parallel SABUL, the thread dealing with both data and control is called the master thread, while the other threads only dealing with data are called slave threads. The master thread must present in SABUL, while slave threads are optional and the number of slave threads can be determined by the number of processors in the system. A parallel SABUL that has only a master thread is back to original SABUL. However, in such a situation its performance will be a little lower because of the synchronization overhead.

Parallel SABUL can be achieved through some modifications to the specification defined in chapter 5.

On the sender side, parallel SABUL simply removes the operations to the control channel from the slave threads, while the master thread is similar to the original SABUL, except for the data packing. Below is the algorithm for the master thread of the sender side.

Data Structure:

1. The application/protocol buffer is linked list with its nodes are the head pointer of the application buffer.
2. The loss sequence number queue is a FIFO queue that stores the loss sequence number from ERR feedback.

Algorithm:

STEP 1. Initialization:
STEP 2. Get current time;
STEP 3. Poll Control Channel. Receive control packet if there is one. Receive and process any feedback. Otherwise, calculate the time passed since last ACK or ERR packet was received, if the interval is greater than the time expired interval, generate an EXP packet and process it.
STEP 4. If the loss queue is not empty, read (and remove) a sequence number and find its position in sender buffer, pack a data packet. GOTO STEP 6;
STEP 5. Start atomic transaction;
  a. Increase the current sequence number; if it is greater than the maximum sequence number, set it to zero. Find next data sending position, if found, pack a data packet and send out;
  End atomic transaction;
STEP 6. Send the packet constructed in STEP 5/6 out to the data channel;
STEP 7. Get current time and calculate the time passed since STEP 2, if it is less than current sending interval, wait until the time passed to reach the interval. GOTO STEP 2.

Reference:

Appendix A.8

On the receiver side, the situation is a little complex. First of all, the generation of feedback only occurs in the master thread. Second, a receiver thread cannot get an expected slot to store receiving data because several threads are receiving data simultaneously. Each thread should read the sequence number of a

packet, find an accurate slot, and then read the whole packet again. The 3 steps above should be in an atomic transaction. The algorithm is depicted as below:

Data Structure:

1. The protocol buffer is a logically circular buffer for temporally storing and reordering the received packets, which is generally large enough considering the data rate.
2. The sequence number array is a bit array that records the packets with which sequence numbers are received and which are not.

Algorithm:

STEP 1. Initialization:

STEP 2.

a. Get current time;
b. If time passed since the last acknowledgment event, or the user registered buffer has been fulfilled, generate an ACK packet;
c. If time passed since the last error feedback event, and the loss list is not empty, generate an ERR packet;

STEP 3. Start atomic transaction

a. Read the sequence number of the next packet, if the timer is expired, go to STEP 2;
b. Find a data slot in the buffer, if it fails, use the temper data slot;
c. Read the whole packet and store the data in the data slot in b;
End the atomic transaction;

STEP 4. If the offset is greater than or equal to the sequence number array size, generate an ACK packet; if the offset is less than 0, go to STEP 2;

STEP 5. Modify the largest received sequence number;

STEP 6. If the size of the current received packet is less than the regular size, insert the sequence number and size into the irregular packet list;

STEP 7. Set the proper slot in the offset flag, go to STEP 2.

Reference:

Appendix A.9

The generation of feedback should also be changed. Since in parallel SABUL the master thread does not know when packet loss happens, it cannot generate the ERR packet in time. Instead, all ERR packets are generated by scanning the sequence number array activated by a timer. ACK packets generation also needs to scan the flag array.

Operations to shared variables in the sender and receiver threads should be synchronized, especially the operation to the protocol buffer, the generation of sequence numbers on the sender side, and the loss list operation on the sender side. Detailed implementation depends on the platform.

7. File Transmission

The SABUL protocol is designed for huge amounts of data transfer over high-speed networks. The use scenarios of SABUL show that a lot of applications in this situation will send files from storage to networks directly and/or receive and store data directly to storage.

The direct support of file manipulation in the protocol enables the possibility to reduce the overhead of data copy between memory and the file system. SABUL specification defines two APIs for file operation: sendfile() and recvfile().

The parameters of these two functions should include the file name, the offset from which point the data will be read/written, and the expected read/write size.

bool sendfile(const int& fd, const int& offset, const int& size);
long recvfile(const int& fd, const int& offset, const int& size);

Since SABUL is an application level protocol, the implementation of the APIs depends on the platforms, so does the performance gained by the introduction of direct file transmission support. Mapped I/O has been regarded as a standard solution for avoidance of copy between disk file and physical memory. For example, in a POSIX system, SABUL can use a mmap() system call to map a file into virtual memory, and avoid using a read/write system call for explicit data copy between memory and a file system.

8. Application Programming Interface

This chapter lists the application programming interfaces (API) that an implementation of SABUL should provide. These APIs specifications are given in C++ format here. Implementations with other languages, e.g., ANSI C, should provide equivalent functions.

The structure SabulOption is used to set and get SABUL parameters settings.

```
struct SabulOption
{
   int m_iDataPort;
   int m_iCtrlPort;
   int m_iPktSize;
   int m_iPayloadSize;
   int m_iMaxLossLength;
   int m_iBlockable;
   int m_lSendBufSize;
   int m_lRecvBufSize;
   int m_iRecvFlagSize;
};
```

Name	Meaning	Type	Comment
m_iDataPort	Data Port	Integer (0,65535]	Read only
m_iCtrlPort	Control Port	Integer (0,65535]	Read only
m_iPktSize	Regular Packet Size	Integer [0, 1472]	Constant
m_iPayloadSize	Regular Data Packet Payload Size	Integer [0, 1472]	Constant
m_iMaxLossLength	Maximum Length of the Loss List	Integer [0, 357]	Constant
m_iBlockable	If current connection is blocking	Integer [0, 1]	0 is non-blocking, otherwise is blocking
m_lSendBufSize	Sender Buffer Size	32-bit Integer	
m_lRecvBufSize	Receiver Buffer Size	32-bit Integer	
m_iRecvFlagSize	Receiver Side Sequence Number Flag Size	Integer	Read only

```
class CSabulSender
{
public:
   CSabulSender();
   virtual ~CSabulSender();

   int open(const int& port = -1);
   void listen();
   void close();

   bool send(char* data, const long& len) const;
   bool sendfile(const int& fd, const int& offset, const int& size) const;

   void setOpt(const SabulOption& opt);
   void getOpt(SabulOption& opt) const;

   long getCurrBufSize() const;
   bool getCurrStatus() const;
```

}

API name	Parameter List	Comment
int open();	int port = -1	Open a Sender using the given port as Control Port. If default value is user, SABUL will use default Control Port - 7100. The method returns actual Control Port for a receiver to connect to.
void listen();		Called after open() to wait a receiver's connection.
void close();		Close all open connection, restore all variables to the status before open() is called
bool send()	char* data long len	Sends out a buffer started from *data* with size of *len* The method returns true if successful, otherwise returns false, indicating that sender buffer is full Returns immediately in non-blocking mode, otherwise wait until data is sent out successfully or buffer is full
Bool sendfile()	int fd int offset int size	Sends a file indicated by file number *fd* with size of *size*, *offset* must be zero The method returns true if successful, otherwise returns false, indicating that sender buffer is full Returns immediately in non-blocking mode, otherwise wait until data is sent out successfully or buffer is full
void setOpt()	SabulOption& opt	Configure sender with given parameter of *opt*
void getOpt()	SabulOption& opt	Retrieve current configuration into *opt*
long getCurrBufSize()		Retrieve current Sender buffer size Returned value is the size
bool getCurrStatus()		Returns current sender status - sending data or waiting for new data Returned true if sender is sending data, otherwise returns false

```
class CSabulRecver
{
public:
   CSabulRecver();
   virtual ~CSabulRecver();

   int open(const char* ip, const int& port);
   void close();

   long recv(char* data, const long& len);
   long recvfile(const int& fd, const int& offset, const int& size);

   void setOpt(const SabulOption& opt);
   void getOpt(SabulOption& opt) const;
}
```

API name	Parameter List	Comment
int open()	char* ip int port	Open a Receiver and connect to a sender at host with *ip* and *port*

void close()		Close all open connections, restore all variables to the status before open() is called
long recv()	char* data long len	Receives data with expected size of *len* Returns the actual size of received data; Returns -1 if in non-blocking mode and not enough data in buffer to receiver
long recvfile();	int fd int offset int size	Receives data and store in a file with file number of *fd* and expected size of *size*; *offset* must be 0; Returns the actual size of received data; Returns -1 if in non-blocking mode and not enough data in buffer to receiver
void setOpt()	SabulOption& opt	Configure receiver with given parameter of *opt*
void getOpt()	SabulOption& opt	Retrieve current configuration into *opt*

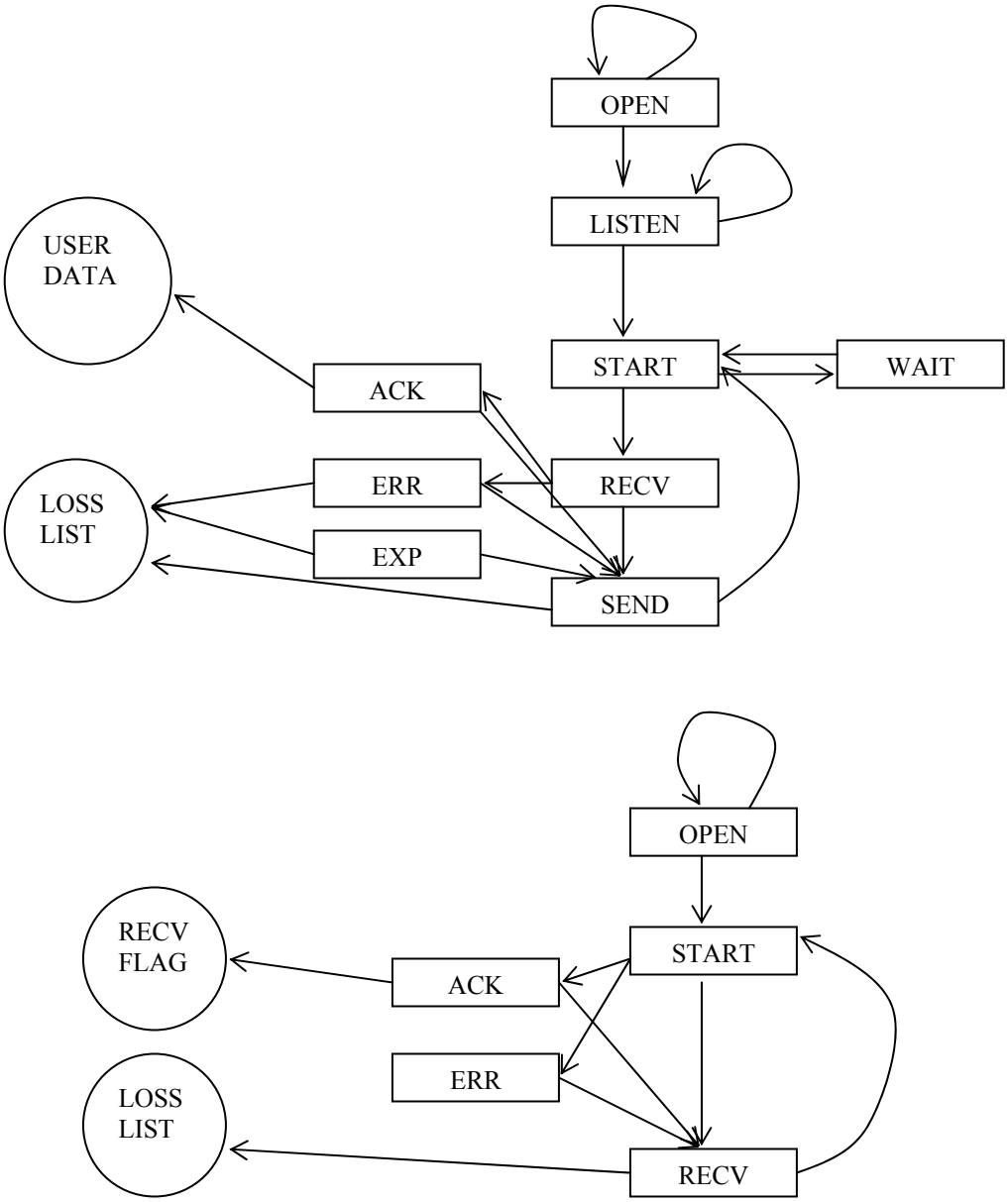
9. Summary of Protocol Constants and Parameters

This chapter contains a summary listing of the constants defined in this specification. The following constants are recommended in Ethernet with MSU of 1500 bytes and more than 100M bandwidth. Applications or implementations running on some other environments should change these values to obtain better performance.

Constant Variable	Value	Comment
Packet Size	1472	
Payload Size	1468	
Maximum Length of Loss List	366	
Default Data Port Number	7000	
Default Control Port Number	7001	
Maximum Sequence Number	2^31	
Sequence Number Valve	2^30	
SYN Interval	10^6 us	
EXP Interval	10^6 us	
ACK Interval	10^5 us	
ERR Interval	20000 us	
Loss Rate Upper Bound	0.01	
Loss Rate Lower Bound	0.001	
Smooth Loss Average weight	0.1	
Initial Data Sending Interval	10 us	
Data Sending Interval Limit	[1, 200] us	The upper limit should be large enough for the give physical network bandwidth
Receiver Sequence Number Flag Size	25600	
Unacknowledged Packet Number Threshold for Rate Control	19200	
Number of Continuously Received ERR Packet for Rate Control	50	
Default Sender Side Buffer Size Limit	40960000	
Default Receiver Side Buffer Size	40960000	
Blocking Mode	false	Default value
Use Rate Acceleration	true	Default value

10. Automaton and Protocol Verification

There are two phases in SABUL life cycle: the connection phase and the transfer phase. The figure below is a transition graph of the SABUL protocol.



Since the connection phase is simple and the relation between the two phases is loose, only the data transfer phase is considered in the model.

Based on the SABUL specification and the transition system described in section 2, SABUL can be modeled in the following rules:

1) Sender:
   a. START -> WAIT iff user data is empty

b. WAIT -> START iff new user data comes
c. START -> RECV iff user data is not empty
d. RECV -> ACK iff ACK packet comes
e. RECV -> ERR iff ERR packet comes
f. RECV -> EXP iff NO ACK/ERR received after a time-out interval since last ACK/ERR received
g. RECV -> SEND iff no event above happened
h. ACK/ERR/EXP -> SEND
i. SEND -> START

Where ACK decreases the user data, ERR increases the loss list, and SEND decreases the loss list if it is not empty.

End state: WAIT

2) Receiver

a. START -> ACK iff ACK timer is expired and there are continuous packets received starting from 0 offset in RECV FLAG
b. START -> ERR iff the ERR timer is expired and the loss list is not empty
c. START -> RECV iff no event above happened
d. ACK/ERR -> RECV
e. RECV -> START iff packet received or timer expired

Where RECV increases the loss list if the current packet is larger and not continuous from the largest sequence number that has been received by now, RECV decreases the loss list if current packet is less than the largest sequence number that has been received by now, and ACK shifts the RECV FLAG.

End state: START

The transition system above can be verified using a model checker such as SPIN, which is developed in Bell Lab and is very suitable for communication protocols.

When model checking SABUL, some properties need to be verified in addition to the deadlock detection. For example:
1. Safety properties:
AG ((sender user data empty) -> (receiver RECV FLAG empty and loss list empty))

2. Liveness properties:
AG (new data added) -> AF (ACK received)

11. Conclusion and Future Work

Rate adjusting UDP has been regarded as an effective way to overcome the poor performance of TCP over high-speed networks. SABUL is such a transport protocol employing this technology.

The basic idea of SABUL is to use UDP to transfer data and exchange the control information over another TCP channel. Since the control information is much less than the user data, the idea is sound and has been successful in real testing.

However, this simple model is not enough for an ideal high performance transport protocol. Such a protocol may transfer data at a speed of several gigabits per second, which intend to use up the bandwidth and the processors time at end systems. We have identified the following problems in the design of such a protocol and put out the solutions in SABUL.

1. What is a rate control algorithm that can utilize all the available bandwidth while also sharing the bandwidth fairly with other coexisting connections?

The answer to this question is highly related to how we define the fairness of network utilization.

Today the TCP is the most popular transport protocol, so a TCP-friendly protocol is generally regarded as a fair protocol. Under this definition of fairness, the rate adjustment of SABUL can be done using the following rule: Calculate the TCP bandwidth according to the current loss rate and then adjust the SABUL sending rate to meet it.

However, we argue that TCP-friendly fairness is not an ideal definition of fairness: It can neither utilize bandwidth efficiently nor reach real fairness. The AIMD algorithm in TCP causes the average transfer speed to be lower than the theoretical speed it can reach; the introduction of RTT makes the connection with greater delay share lower bandwidth than the other connections in the same link.

SABUL provides another rate control algorithm that is based on the loss rate only. It adjusts the sending rate multiple times with a parameter of the current loss rate.

2. How to reduce the memory copy between application and the protocol and the memory copy between the protocol and the operating system?

Most of the modern operating systems copy the data from user space to kernel space when sending data through a network interface, and copy the data from kernel space to user space when receiving. In addition, application level protocols like SABUL often need to maintain their own buffer. The memory copy problem is extremely serious in high performance network protocols. Moving 1G bytes data between two memory blocks may use up the processor time.

In SABUL we have introduced user buffer registration method to reduce in copy from protocol buffer to application. The popular data scatter/gather method is also used in sending/receiving data packets.

3. How the protocol can transparently use multiple processors at the end of extremely high networks?

When transferring data over extremely high networks, say, 10G Ethernet, one single processor in current popular calculation ability, e.g., Intel Pentium 4 2.0G, cannot meet the requirement of data processing (e.g., copy, calculation, etc.) in the end system. Generally, in such a situation, the end system is a cluster or a SMP computer with multiple processors.

A high performance application level protocol should utilize multiple processors transparently. The parallel SABUL is to solve the problem. Parallel SABUL use one master thread to deal with both data and control information, and slave threads in optional numbers to deal with data.

SABUL has partly solved the problems above. In the future, we expect to build SABUL into a system kernel to eliminate the memory copy between user space and kernel space, and to deploy a high precision timer.

Currently parallel SABUL is still being tested. We have not used it in any real applications. The parallel SABUL is expected to gain satisfied performance in those LANs, WANs and SANs where network bandwidth is extremely high and the end systems are regular cluster or multi-processor computer.

Below is a list of the SABUL products that have been done, is being done or will be done.

1	√	SABUL Protocol version 2.0 *"socket" like API, efficient rate control, reliable data transfer, direct file support*	Apr. 2001
2	√	Parallel SABUL transparent multiple CPUs support	May 2002
3		Theoretical Research of Rate Control over High Speed Networks *efficient, TCP-friendly rate control* *router's effect on UDP rate control*	Aug. 2002
4		Fast SABUL fast by permitting limited packet loss	Aug. 2002
5		Hardware Timer and Zero Copy UDP socket API *high precision timer* *copy avoidance in kernel level*	Nov. 2002
6		Porting SABUL into Kernel (SABUL 3.0) efficient, TCP-friendly rate control, zero memory copy	Jan. 2003

Appendix A. Algorithms

Important algorithms in the SABUL sender and receiver are presented here by C++ program examples. These implementation notes are for informational purposes only and are meant to clarify the SABUL specification.

These C++ programs work on the Linux system on the Intel IA-32 architecture. There may be other implementation methods that are faster in particular operating environments or have other advantages.

A.1 Rate Control Algorithm

```
void CSabulSender::rateControl(const double& currlossrate)
{
  m_dHistoryRate = m_dHistoryRate * m_dWeight + currlossrate * (1 - m_dWeight);

  if (m_bAcceleration)
  {
    if (m_dHistoryRate > m_dRateUpperBound)
    {
      m_bAcceleration = false;
      m_iInterval <<= 1;
    }
    else if (m_dHistoryRate < m_dRateLowerBound)
    {
      m_iInterval >>= 1;
      if (m_iInterval <= 1)
      {
        m_bAcceleration = false;
        m_iInterval = 1;
      }
    }
    else
      m_bAcceleration = false;

    return;
  }

  if (m_dHistoryRate > m_dRateUpperBound)
  {
    if (m_iInterval == int(m_iInterval * (1. + (m_dHistoryRate - m_dRateUpperBound))))
      m_iInterval ++;
    else
      m_iInterval = int(m_iInterval * (1. + (m_dHistoryRate - m_dRateUpperBound)));
    if (m_iInterval > m_iThreshold)
      m_iInterval = m_iThreshold;
  }
  else if (m_dHistoryRate < m_dRateLowerBound)
  {
    m_iInterval = int(m_iInterval * (1. -  (m_dRateLowerBound - m_dHistoryRate)));
    if (m_iInterval < 1)
      m_iInterval = 1;
  }
}
```

A.2 Feedback Processing in Sender Side

```
void CSabulSender::processFeedback(const PktType type, const long attr, const long* data)
{
  switch (type)
  {
  case SC_ACK:
    if (attr >= m_lLastAck)
      m_pBuffer->ackData((attr - m_lLastAck) * m_iPayloadSize, m_iPayloadSize);
    else
      m_pBuffer->ackData((attr - m_lLastAck + m_lMaxSeqNo) * m_iPayloadSize,
m_iPayloadSize);
    m_lLastAck = attr;
    m_pLossList->remove(m_lLastAck);

    break;

  case SC_SYN:
    m_lLocalSend += m_lCurrSeqNo - m_lLastSYNSeqNo;
    if (m_lCurrSeqNo < m_lLastSYNSeqNo)
      m_lLocalSend += m_lMaxSeqNo;

    if (m_lLocalSend != 0)
    {
      cout<< "loss rate "<< double(m_lLocalSend - attr) / double(m_lLocalSend) << endl;
      rateControl(double(m_lLocalSend - attr) / double(m_lLocalSend));
      m_lLocalSend = 0;
      cout<< "RATE CONTROL: "<< m_iInterval<< endl;
      m_ullInterval = m_iInterval * m_ullCPUFrequency;
    }

    m_lLastSYNSeqNo = m_lCurrSeqNo;
```

```
    m_lLocalSend = 0;
    m_iERRCount = 0;

    break;

  case SC_ERR:
    if ((attr != m_iMaxLossLength) && (++ m_iERRCount == 50))
    {
      m_iInterval += 2;
      m_ullInterval += 2 * m_ullCPUFrequency;
      m_iERRCount = -200;
    }

    for (int i = attr - 1; (i >= 0) && (m_pLossList->insert(data[i])); i --) {}

    break;

  case SC_EXP:
    for (int i = m_lCurrSeqNo - m_lLastAck - 1; (i >= 0) && (m_pLossList->insert(i +
m_lLastAck)); i --) {}
  }
}
```

A.3 Packet Sending Algorithm

```
void* CSabulSender::dcHandler(void* sender)
{
  CSabulSender* self = static_cast<CSabulSender *>(sender);

  CPacketVector sdpkt(self->m_iPktSize);
  int payload;
  long offset;

  CPktSblCtrl scpkt(self->m_iPktSize);

  unsigned long long int entertime;
  unsigned long long int nextexptime;
  self->rdtsc(nextexptime);

  unsigned long long int ullexpint = self->m_iEXPInterval * self->m_ullCPUFrequency;

  nextexptime += ullexpint;

  while (!self->m_bClosing)
  {
    self->rdtsc(entertime);

    if (entertime > nextexptime)
    {
      self->processFeedback(SC_EXP);
      nextexptime = entertime + ullexpint;
    }

    if (0 == self->m_pBuffer->getCurrBufSize())
    {
      pthread_mutex_lock(&(self->m_SendDataLock));
      if (0 == self->m_pBuffer->getCurrBufSize())
        pthread_cond_wait(&(self->m_SendDataCond), &(self->m_SendDataLock));
      pthread_mutex_unlock(&(self->m_SendDataLock));

      self->rdtsc(nextexptime);
      nextexptime += ullexpint;
    }

    *(CTcpChannel *)(self->m_pCtrlChannel) >> scpkt;
    if (scpkt.getLength() > 0)
    {
      self->processFeedback(PktType(scpkt.m_lPktType), scpkt.m_lAttr, scpkt.m_plData);
      if (scpkt.m_lPktType != SC_SYN)
      {
        self->rdtsc(nextexptime);
        nextexptime += ullexpint;
      }
    }

    if ((sdpkt.m_lSeqNo = self->m_pLossList->getLostSeq()) >= 0)
    {
      if ((sdpkt.m_lSeqNo >= self->m_lLastAck) && (sdpkt.m_lSeqNo < self->m_lLastAck +
self->m_lSeqNoTH))
        offset = (sdpkt.m_lSeqNo - self->m_lLastAck) * self->m_iPayloadSize;
      else if (sdpkt.m_lSeqNo < self->m_lLastAck - self->m_lSeqNoTH)
        offset = (sdpkt.m_lSeqNo + self->m_lMaxSeqNo - self->m_lLastAck) * self-
>m_iPayloadSize;
      else
        continue;

      if ((payload = self->m_pBuffer->readData(&(sdpkt.m_pcData), offset, self-
>m_iPayloadSize)) == 0)
        continue;

      self->m_lLocalSend ++;
    }
```

```
    else
    {
      if (self->m_lCurrSeqNo - self->m_lLastAck >= self->m_RemoteOpt.m_iRecvFlagSize)
        continue;

      if ((payload = self->m_pBuffer->readData(&(sdpkt.m_pcData), self->m_iPayloadSize)) ==
0)
        continue;

      sdpkt.m_lSeqNo = self->m_lCurrSeqNo ++;
      self->m_lCurrSeqNo %= self->m_lMaxSeqNo;
    }

    sdpkt.setLength(payload);
    *(CUdpChannel *)(self->m_pDataChannel) << sdpkt;

    self->sleepto(entertime + self->m_ullInterval);
  }

  return NULL;
}
```

A.4 Generation of Feedback on the Receiver Side

```
inline void CSabulRecver::feedback(PktType type, const int& len, const long* data)
{
  CPktSblCtrl scpkt(m_iPktSize);

  switch (type)
  {
  case SC_ACK:
    if (0 == m_pLossList->getLossLength())
    {
      if (m_lCurrSeqNo >= m_lLastAck)
        scpkt.m_lAttr = m_lCurrSeqNo - m_lLastAck + 1;
      else if (m_lLastAck - m_lCurrSeqNo > m_lSeqNoTH)
        scpkt.m_lAttr = m_lCurrSeqNo + m_lMaxSeqNo - m_lLastAck + 1;
    }
    else
      scpkt.m_lAttr = (m_pLossList->getFirstLostSeq() >= m_lLastAck) ? m_pLossList-
>getFirstLostSeq() - m_lLastAck : m_pLossList->getFirstLostSeq() + m_lMaxSeqNo -
m_lLastAck;

    if (0 < scpkt.m_lAttr)
    {
      m_RecvFlag >>= scpkt.m_lAttr;
      m_lNextExpect -= scpkt.m_lAttr;
      m_lLastAck = (m_lLastAck + scpkt.m_lAttr) % m_lMaxSeqNo;
      if (m_pBuffer->ackData(scpkt.m_lAttr * m_iPayloadSize - m_pIrregularPktList-
>currErrorSize(m_lLastAck)))
      {
        pthread_mutex_lock(&m_RecvDataLock);
        pthread_cond_signal(&m_RecvDataCond);
        pthread_mutex_unlock(&m_RecvDataLock);
      }
      m_pIrregularPktList->deleteIrregularPkt(m_lLastAck);

      scpkt.m_lPktType = SC_ACK;
      scpkt.m_lAttr = m_lLastAck;
      scpkt.setLength(2 * sizeof(long));
      *(CTcpChannel *)(m_pCtrlChannel) << scpkt;
    }

    break;

  case SC_ERR:
    if (len > 0)
    {
      scpkt.m_lAttr = len;
      memcpy(scpkt.m_plData, data, len * sizeof(long));
    }
    else
    {
      m_pLossList->getLossArray((long *)(scpkt.m_plData), int(scpkt.m_lAttr),
m_iMaxLossLength, m_iRTT);

      if (0 == scpkt.m_lAttr)
        break;
    }

    scpkt.m_lPktType = SC_ERR;
    scpkt.setLength((scpkt.m_lAttr + 2) * sizeof(long));
    *(CTcpChannel *)(m_pCtrlChannel) << scpkt;

    break;

  case SC_SYN:
    scpkt.m_lPktType = SC_SYN;
    scpkt.m_lAttr = m_lLocalRecv;
    scpkt.setLength(2 * sizeof(long));

    m_lLocalRecv = 0;
```

```
    *(CTcpChannel *)(m_pCtrlChannel) << scpkt;
  }
}
```

A.5 Calculating next expected sequence number

```
inline void CSabulRecver::getNextExpect(const long& offset)
{
  m_lNextExpect = offset + 1;
  m_lNextExpect %= FlagSize;
  while (m_RecvFlag[m_lNextExpect])
  {
    m_lNextExpect ++;
    m_lNextExpect %= FlagSize;
    if (m_lNextExpect == offset)
    {
      feedback(SC_ACK);
      m_lNextExpect = FlagSize;
    }
  }
}
```

A.6 User Buffer Registration

```
void CSabulRecver::CBuffer::registerUserBuf(char* buf, const long& len)
{
  m_lUserBufAck = 0;
  m_lUserBufSize = len;
  m_pcUserBuf = buf;
  m_bAfterUserBufBoundary = false;

  long currwritepos = (m_lLastAckPos + m_lMaxOffset) % m_lSize;

  if (m_lStartPos < currwritepos)
    if (currwritepos - m_lStartPos <= len)
    {
      memcpy(m_pcUserBuf, m_pcData + m_lStartPos, currwritepos - m_lStartPos);
      m_lMaxOffset = 0;
    }
    else
    {
      memcpy(m_pcUserBuf, m_pcData + m_lStartPos, len);
      m_lMaxOffset -= len - (m_lLastAckPos- m_lStartPos);
    }
  else if (m_lStartPos > currwritepos)
    if (m_lSize - (m_lStartPos - currwritepos) <= len)
    {
      memcpy(m_pcUserBuf, m_pcData + m_lStartPos, m_lSize - m_lStartPos);
      memcpy(m_pcUserBuf + m_lSize - m_lStartPos, m_pcData, currwritepos);
      m_lMaxOffset = 0;
    }
    else
    {
      if (m_lSize - m_lStartPos <= len)
      {
        memcpy(m_pcUserBuf, m_pcData + m_lStartPos, m_lSize - m_lStartPos);
        memcpy(m_pcUserBuf + m_lSize - m_lStartPos, m_pcData, len - (m_lSize -
m_lStartPos));
      }
      else
        memcpy(m_pcUserBuf, m_pcData + m_lStartPos, len);
      m_lMaxOffset -= len - (m_lLastAckPos + m_lSize - m_lStartPos);
    }

  if (m_lStartPos < m_lLastAckPos)
    m_lUserBufAck += m_lLastAckPos - m_lStartPos;
  else if (m_lStartPos > m_lLastAckPos)
    m_lUserBufAck += m_lSize - m_lStartPos + m_lLastAckPos;

  m_lStartPos = (m_lStartPos + len) % m_lSize;
  m_lLastAckPos = m_lStartPos;
}
```

A.7 Packet Receiving Algorithm

```
void* CSabulRecver::dcHandler(void* recver)
{
  CSabulRecver* self = static_cast<CSabulRecver *>(recver);
  CPacketVector sdpkt(self->m_iPayloadSize);
  char payload[self->m_iPayloadSize];
  bool nextslotfound;
  long offset;
  long lossdata[self->m_iMaxLossLength];

  unsigned long long int currtime;
  unsigned long long int nextacktime;
  unsigned long long int nexterrtime;
  unsigned long long int nextsyntime;
  unsigned long long int ullackint = self->m_iACKInterval * self->m_ullCPUFrequency;
  unsigned long long int ullerrint = self->m_iERRInterval * self->m_ullCPUFrequency;
  unsigned long long int ullsynint = self->m_iSYNInterval * self->m_ullCPUFrequency;
```

```
  self->rdtsc(nextacktime);
  nextacktime += ullackint;
  self->rdtsc(nextsyntime);
  nextsyntime += ullsynint;

  while (!self->m_bClosing)
  {
    if (self->m_bReadBuf)
    {
      self->m_bReadBuf = self->m_pBuffer->readBuffer(const_cast<char*>(self->m_pcTempData), const_cast<long&>(self->m_lTempLen));
      if (!self->m_bReadBuf)
        self->m_pBuffer->registerUserBuf(const_cast<char*>(self->m_pcTempData), const_cast<long&>(self->m_lTempLen));
      else
      {
        self->m_bReadBuf = false;
        pthread_mutex_lock(&(self->m_RecvDataLock));
        pthread_cond_signal(&(self->m_RecvDataCond));
        pthread_mutex_unlock(&(self->m_RecvDataLock));
      }
    }

    self->rdtsc(currtime);
    if ((currtime > nextacktime) || self->m_pBuffer->reachUserBufBoundary())
    {
      self->feedback(SC_ACK);
      nextacktime = currtime + ullackint;
    }
    if ((currtime > nexterrtime) && (self->m_pLossList->getLossLength() > 0))
    {
      self->feedback(SC_ERR);
      nexterrtime = currtime + ullerrint;
    }
    if (currtime > nextsyntime)
    {
      self->feedback(SC_SYN);
      nextsyntime = currtime + ullsynint;
    }

    sdpkt.setLength(self->m_iPayloadSize);

    if (!(self->m_pBuffer->nextDataPos(&(sdpkt.m_pcData), self->m_lNextExpect * self->m_iPayloadSize - self->m_pIrregularPktList->currErrorSize(self->m_lNextExpect + self->m_lLastAck), self->m_iPayloadSize)))
    {
      sdpkt.m_pcData = payload;
      nextslotfound = false;
    }
    else
      nextslotfound = true;

    *(CUdpChannel *)(self->m_pDataChannel) >> sdpkt;

    if (sdpkt.getLength() <= 0)
      continue;

    self->m_lLocalRecv ++;

    offset = sdpkt.m_lSeqNo - self->m_lLastAck;

    if (offset >= FlagSize)
    {
      self->feedback(SC_ACK);
      self->rdtsc(nextacktime);
      nextacktime += ullackint;
      continue;
    }
    else if (offset < 0)
      if (offset < -self->m_lSeqNoTH)
        offset += self->m_lMaxSeqNo;
      else
        continue;

    if ((offset != self->m_lNextExpect) || (!nextslotfound))
    {
      if (!(self->m_pBuffer->addData(sdpkt.m_pcData, offset * self->m_iPayloadSize - self->m_pIrregularPktList->currErrorSize(sdpkt.m_lSeqNo), sdpkt.getLength()));
        continue;

      if (offset > self->m_lNextExpect)
      {
        int losslen = 0;
        int start = (self->m_lCurrSeqNo + 1 >= self->m_lLastAck) ? (self->m_lCurrSeqNo - self->m_lLastAck + 1) : (self->m_lCurrSeqNo + self->m_lMaxSeqNo - m_lLastAck;
        for (int i = start; i < offset; i ++)
          if (!self->m_RecvFlag[i])
          {
            self->m_pLossList->insert((i + self->m_lLastAck) % self->m_lMaxSeqNo);
            if (losslen < self->m_iMaxLossLength)
              lossdata[losslen ++] = (i + self->m_lLastAck) % self->m_lMaxSeqNo;
```

```
            }
          self->feedback(SC_ERR, losslen, lossdata);
        }
      }

      if (sdpkt.getLength() != self->m_iPktSize)
        self->m_pIrregularPktList->addIrregularPkt(sdpkt.m_lSeqNo, self->m_iPktSize -
sdpkt.getLength());

      if ((sdpkt.m_lSeqNo > self->m_lCurrSeqNo) && (sdpkt.m_lSeqNo - self->m_lCurrSeqNo <
self->m_lSeqNoTH))
        self->m_lCurrSeqNo = sdpkt.m_lSeqNo;
      else if (sdpkt.m_lSeqNo < self->m_lCurrSeqNo - self->m_lSeqNoTH)
        self->m_lCurrSeqNo = sdpkt.m_lSeqNo;
      else
      {
        self->m_pLossList->remove(sdpkt.m_lSeqNo);

        if (sdpkt.getLength() < self->m_iPktSize)
          self->m_pBuffer->moveData((offset + 1) * self->m_iPayloadSize - self-
>m_pIrregularPktList->currErrorSize(sdpkt.m_lSeqNo), self->m_iPktSize - sdpkt.getLength());
      }

      self->getNextExpect(offset);
      self->m_RecvFlag.set(offset);
    }

    return NULL;
}
```

A.8 Parallel Sender

```
void* CSabulSender::master(void* sender)
{
  CSabulSender* self = static_cast<CSabulSender *>(sender);

  CPacketVector sdpkt(self->m_iPktSize);
  int payload;
  long offset;

  CPktSblCtrl scpkt(self->m_iPktSize);

  unsigned long long int entertime;
  unsigned long long int nextexptime;
  self->rdtsc(nextexptime);

  unsigned long long int ullexpint = self->m_iEXPInterval * self->m_ullCPUFrequency;

  nextexptime += ullexpint;

  while (!self->m_bClosing)
  {
    self->rdtsc(entertime);

    if (entertime > nextexptime)
    {
      self->processFeedback(SC_EXP);
      nextexptime = entertime + ullexpint;
    }

    if (0 == self->m_pBuffer->getCurrBufSize())
    {
      pthread_mutex_lock(&(self->m_SendDataLock));
      if (0 == self->m_pBuffer->getCurrBufSize())
        pthread_cond_wait(&(self->m_SendDataCond), &(self->m_SendDataLock));
      pthread_mutex_unlock(&(self->m_SendDataLock));

      self->rdtsc(nextexptime);
      nextexptime += ullexpint;
    }

    *(CTcpChannel *)(self->m_pCtrlChannel) >> scpkt;
    if (scpkt.getLength() > 0)
    {
      self->processFeedback(PktType(scpkt.m_lPktType), scpkt.m_lAttr, scpkt.m_plData);
      if (scpkt.m_lPktType != SC_SYN)
      {
        self->rdtsc(nextexptime);
        nextexptime += ullexpint;
      }
    }

    if ((sdpkt.m_lSeqNo = self->m_pLossList->getLostSeq()) >= 0)
    {
      if ((sdpkt.m_lSeqNo >= self->m_lLastAck) && (sdpkt.m_lSeqNo < self->m_lLastAck +
self->m_lSeqNoTH))
        offset = (sdpkt.m_lSeqNo - self->m_lLastAck) * self->m_iPayloadSize;
      else if (sdpkt.m_lSeqNo < self->m_lLastAck - self->m_lSeqNoTH)
        offset = (sdpkt.m_lSeqNo + self->m_lMaxSeqNo - self->m_lLastAck) * self-
>m_iPayloadSize;
      else
        continue;
```

```
      payload = self->m_pBuffer->readData(&(sdpkt.m_pcData), offset, self->m_iPayloadSize);

      if (0 == payload)
        continue;
    }
    else
    {
      if (self->m_lCurrSeqNo - self->m_lLastAck >= self->m_RemoteOpt.m_iRecvFlagSize)
        continue;

      pthread_mutex_lock(&(self->m_NewPktLock));
      payload = self->m_pBuffer->readData(&(sdpkt.m_pcData), self->m_iPayloadSize);
      if (0 == payload)
      {
        pthread_mutex_unlock(&(self->m_NewPktLock));
      }
      sdpkt.m_lSeqNo = self->m_lCurrSeqNo ++;
      self->m_lCurrSeqNo %= self->m_lMaxSeqNo;
      pthread_mutex_unlock(&(self->m_NewPktLock));
    }

    sdpkt.setLength(payload);
    *(CUdpChannel *)(self->m_pDataChannel) << sdpkt;

    self->m_lLocalSend ++;

    self->sleepto(entertime + self->m_ullInterval);
  }

  return NULL;
}
```

A.9 Parallel Receiver

```
void* CSabulRecver::master(void* recver)
{
  CSabulRecver* self = static_cast<CSabulRecver *>(recver);
  CPacketVector sdpkt(self->m_iPayloadSize);

  int offset;

  unsigned long long int currtime;
  unsigned long long int nextacktime;
  unsigned long long int nexterrtime;
  unsigned long long int nextsyntime;
  unsigned long long int ullackint = self->m_iACKInterval * self->m_ullCPUFrequency;
  unsigned long long int ullerrint = self->m_iERRInterval * self->m_ullCPUFrequency;
  unsigned long long int ullsynint = self->m_iSYNInterval * self->m_ullCPUFrequency;

  self->rdtsc(nextacktime);
  nextacktime += ullackint;
  self->rdtsc(nextsyntime);
  nextsyntime += ullsynint;

  while (!self->m_bClosing)
  {
    if (self->m_bReadBuf)
    {
      self->m_bReadBuf = self->m_pBuffer->readBuffer(const_cast<char*>(self-
>m_pcTempData), const_cast<long&>(self->m_lTempLen));
      if (!self->m_bReadBuf)
        self->m_pBuffer->registerUserBuf(const_cast<char*>(self->m_pcTempData),
const_cast<long&>(self->m_lTempLen));
      else
      {
        self->m_bReadBuf = false;
        pthread_mutex_lock(&(self->m_RecvDataLock));
        pthread_cond_signal(&(self->m_RecvDataCond));
        pthread_mutex_unlock(&(self->m_RecvDataLock));
      }
    }

    self->rdtsc(currtime);
    if ((currtime > nextacktime) || self->m_pBuffer->reachUserBufBoundary())
    {
      self->feedback(SC_ACK);
      nextacktime = currtime + ullackint;
    }
    if (currtime > nexterrtime)
    {
      self->feedback(SC_ERR);
      nexterrtime = currtime + ullerrint;
    }
    if (currtime > nextsyntime)
    {
      self->feedback(SC_SYN);
      nextsyntime = currtime + ullsynint;
    }

    offset = self->recvPkt(sdpkt);
    if (offset < 0)
      continue;
```

```
    if (sdpkt.getLength() != self->m_iPktSize)
    {
      pthread_mutex_lock(&(self->m_IrrPktLock));
      self->m_pIrregularPktList->addIrregularPkt(sdpkt.m_lSeqNo, self->m_iPktSize -
sdpkt.getLength());
      pthread_mutex_unlock(&(self->m_IrrPktLock));
      self->m_pBuffer->moveData((offset + 1) * self->m_iPayloadSize - self-
>m_pIrregularPktList->currErrorSize(sdpkt.m_lSeqNo), self->m_iPktSize - sdpkt.getLength();
    }

    self->m_RecvFlag.set(offset);

    if (offset > self->m_iMaxOffset)
      self->m_iMaxOffset = offset;
  }

  return NULL;
}
```

Appendix B. Copyright

COPYRIGHT © 2001, 2002, UNIVERSITY OF ILLINOIS. ALL RIGHTS RESERVED.

This software is distributed to individuals for personal non-commercial use and to non-profit entities for non-commercial purposes only. Permission is also granted to commercial enterprises to use this software for evaluation purposes only. It is licensed on a non-exclusive basis, free of charge for these uses.

All parties interested in any other use of the software should contact the Laboratory for Advanced Computing [info@lac.uic.edu]. The license is subject to the following conditions:

1. No support will be provided by the developers or by University of Illinois.
2. Redistribution is not permitted. University of Illinois will maintain a copy of this software as a directly downloadable file, available under the terms of license specified in this agreement.
3. All advertising materials mentioning features or use of this software must display the following acknowledgment: "This product includes software developed by University of Illinois, Chicago, Illinois and its contributors."
4. Neither the name of the University nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY UNIVERSITY OF ILLINOIS AT CHICAGO, AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL UNIVERSITY OF ILLINOIS OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTIONS) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE), PRODUCT LIABILITY, OR OTHERWISE ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Appendix C. Credits

Author

Yunhong Gu [yunhong@lac.uic.edu]

Developers and Development History

Version 1.0 - Harinath Sivakumar, Marco Mazzucco, Yiting Pan, and Qiao Zhang
First prototype of SABUL was developed during Spring 2001. This version introduces the basic idea of using rate adjusting UDP to transfer bulk data and TCP to transfer control information. It uses a fixed size buffer and data can be only transfer buffer by buffer. Version 1 calculates loss rate on sender side. The receiver sends back an END acknowledgment after it receives all the data for one buffer. Two threads are used for data and control separately.

Details of version 1 can be found in paper: H. Sivakumar, M. Mazzucco, Y. Pan, Q. Zhang and R. Grossman, Simple available bandwidth utilization library for high speed wide area networks.

Version 1.1 - Xinwei Hong, and Yunhong Gu
Xinwei and Gu cooperated to improve SABUL in Fall 2001. Version 1.1 can send two fixed size buffer at the same time so that sender can continuously sends out data during it finished the first buffer and waiting for the acknowledgment. Version 1.1 calculates loss rate on receiver side and sends back an SYN packets to tell sender the loss rate. It also kept a timer for time out event on receiver side and will send back an EXP packet to sender in the case of time out. Version 1.1 provides explicit data copy avoidance in receiver side by using user buffer directly.

Version 2.0a - Xinwei Hong
Xinwei continued the work on SABUL in Spring 2002. The improvement done by him to version 1.1 is NOT included in this document.

Details of version 1.1 and 2.0a can be found in the paper: X. Hong, M. Mazzucco, Y. Gu, H. Sivakumar, R. Grossman: dSABUL: a new protocol for fast data transmission on extremely high speed networks.

Version 2.0b - Yunhong Gu
Gu modified version 1.1 in Spring 2002 during his work of integrating SABUL to FTP and DSTP. Real applications enlighten new improvements for the SABUL protocol, including:

a. Remove the limitation of buffer size and number;
b. Use one thread instead of two on both sides to reduce the synchronization overhead;
c. Add transparent data copy avoidance in the receiver side;
d. Add direct file transfer support;
e. Positive acknowledgment is activated by timer instead of by the end of the buffer block;
f. Move the expired timer to sender side (hence, remove the use of EXP packets);
g. Add the sequence number overflow solution;
h. The a Immediate loss feedback;
i. Immediate positive acknowledgment feedback such that SABUL can send real time data;
j. Introduce new parallel SABUL supporting multiple processors.

Acknowledgment

The development of SABUL is indebted to many other members' work in LAC, especially,
Dr. Marco Mazzucco, for his technical help and management in the development of SABUL;
Ms. Shirley Connelly, for her effort in the revision of this document;
Mr. Scott Wahlstrom, for his assistance to give us an excellent development and testing environment.

Appendix D. Bibliography

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

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

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

[4] N. Seddigh, B. Nandy, and P. Pieda: Study of TCP and UDP interaction for the AF PHB, *Internet Draft*, 1999.

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

[6] Dina Katabi, Mark Hardley, Charlie Rohrs: Internet Congestion Control for Future High Bandwidth-Delay Product Environments, SIGCOMM 2002

[7] Milan Vojnovic, Jean-Yves Le Boundec: On the Long-Run Behavior of Equation-Based Rate Control, SIGCOMM 2002

[8] Aditya Akella, Richard Karp, Christos Papadimitriou, Srinivasan Sesham, Scott Shenker: Selfish Behavior and Stability of the Internet: A Game-Theoretic Analysis of TCP, SIGCOMM 2002

[9] Deepak Bansal, Hari Balakrishnan: Binomial Congestion Control Algorithms, IEEE INFOCOM 2001

[10] S. Fredj, T. Bonald, A. Proutiere, G. Regnie, J. Roberts: Statistical Bandwidth Sharing: A Study of Congestion at Flow Level, SIGCOMM 2001

[11] Deepak Bansak, Hari Balakrishnan, Sally Floyd, Scott Shenker: Dynamic Behavior of Slowly-Responsive Congestion Control Algorithms, SIGCOMM 2001

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

[13] F. Kelly, A. Maulloo, D. Tan: Rate Control for Communication Networks: Shadow Prices, Proportional Fairness and Stability

[14] K. Pamakrishnan, S. Floyd: Proposal to Add Explicit Congestion Notification (ECN) to IP, RFC 2481, Jan. 1999

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

[16] S. Mascolo, C. Casetti, M. Gerla, S. S. Lee, M. Sanadidi: TCP Westwood: Congestion Control with Faster Recovery, UCLA CSD Technical Report #200017, 2000

[17] Mark Allman, Vern Paxson: On Estimating End-to-End Network Path Properties, SIGCOMM 1999

[18] H. Sivakumar S. Bailey R. L. Grossman: PSockets: The Case for Application-level Network Striping for Data Intensive Applications using High Speed Wide Area Networks. Supercomputing 1999

[19] Sally Floyd, Van Jacobson: Random Early Detection Gateways for Congestion Avoidance, IEEE/ACM Transactions on Networking, 1993

[20] H. Schulzrinne, S. Casner, R. Frederick, V. Jacobson: RTP: A Transport Protocol for Real-Time Applications. RFC 1893, Jan. 1996