

Improved Selective Acknowledgment Scheme for TCP

Rajkumar Kettimuthu and William Allcock

Argonne National Laboratory, Globus Alliance

Argonne, IL 60439, USA

{kettimut, allcock}@mcs.anl.gov

Abstract

A selective acknowledgment (SACK) mechanism, combined with a selective repeat retransmission policy, has been proposed to overcome the limitations with the cumulative acknowledgment scheme in TCP. With the SACK mechanism, the receiver informs the sender about the non-contiguous blocks of data that have been received and queued. However, for each such noncontiguous block, SACK requires 8 bytes to convey the information to the sender. Since TCP options field has a fixed length, an acknowledgment packet, at the maximum, can carry information about only 4 noncontiguous blocks. Under some error conditions, this limitation can cause the TCP sender to retransmit packets that have already been received successfully by the receiver. In this paper, we propose an improved selective acknowledgment (ISACK) scheme to overcome the limitations of the current selective acknowledgment scheme. Using examples, we demonstrate how the proposed scheme works. We further propose an adaptive selective acknowledgment (ASACK) strategy that dynamically switches between SACK and ISACK to give optimal performance.

1 Introduction

TCP was originally defined in RFC 793 [15], and several enhancements have been proposed to TCP since then [1, 6, 11]. Recently, researchers have formulated numerous other approaches [5, 8, 9, 12, 13] to address the limitations of the AIMD-based (Additive Increase Multiplicative Decrease) TCP's congestion control algorithm [1] in long-fat networks (networks with large bandwidth and long delay). In this work, we focus on efficiently transferring information about the current state of the receiving TCP to the sending TCP to help the congestion control algorithm at the sending side. In the widely known TCP implementations such as TCP Reno [1] and TCP New-Reno [6], when multiple packets are lost from a window of data, TCP may end up either retransmitting packets that might have already been successfully received or retransmitting at most one dropped packet per round-trip time. In order to overcome this limitation, a selective acknowledgment (SACK) mechanism was defined in RFC 2018 [14]. In TCP SACK, the receiver can inform the sender about all the segments that have been received successfully, allowing the sender to retransmit only the segments that have actually been

lost. SACK uses two 32-bit unsigned integers to represent a noncontiguous block of contiguous data. Hence, SACK needs 8 bytes to convey information about one such block. This 8-byte information is referred to as a "SACK block." The maximum size allowed for the TCP options in a segment is 40 bytes. Apart from the SACK blocks, SACK needs 1 byte each to indicate the kind and the length of the option. The maximum number of SACK blocks that can be carried by an acknowledgment packet is restricted to 4 (the number of bytes needed to represent 4 SACK blocks is 34 or $4 \times 8 + 2$). If the TCP timestamp option [11] is used, as is typical, the maximum number of SACK blocks is reduced to 3 (the timestamp option requires 10 bytes, leaving only 30 bytes for the SACK option). This restriction can sometimes cause unnecessary retransmission of successfully received packets. The result is an unnecessary reduction in the TCP congestion window and a decrease in the throughput of the TCP connections.

In this paper, we propose an improved selective acknowledgment (ISACK) mechanism to overcome this limitation. We further propose an adaptive selective acknowledgment (ASACK) scheme that uses both SACK and ISACK to get better performance. In Section 2, we provide background on TCP and its congestion control algorithms. In Section 3, we provide a brief introduction to TCP SACK and elaborate on its limitation. In Section 4, we describe our ISACK scheme and demonstrate it with an example. In Section 5, we describe the ASACK scheme; and in Section 6, we summarize our results.

2 Background

TCP provides connection-oriented, reliable byte stream service. When two processes wish to communicate, their TCPs must first establish a connection; in other words, they must send some preliminary segments to each other to establish the parameters of the ensuing data transfer. TCP ensures that the data arrives undamaged and in order (no data is lost, none is repeated, and none is subject to error in transmission). TCP provides this reliability by assigning a sequence number to each octet transmitted and by requiring a positive acknowledgment (ACK) from the receiving TCP. The acknowledgment mechanism is cumulative: an acknowledgment of sequence number X indicates that all octets up to but not including X have been received. If the ACK is not received within a timeout interval, the data is

retransmitted. At the receiver, the sequence numbers are used to order segments that may be received out of order and to eliminate duplicates. Damage is handled by adding a checksum to each segment transmitted, checking it at the receiver and discarding damaged segments.

The service provided by TCP is called byte stream because an application that uses the TCP service is unaware of the fact that data is broken into segments for transmission over the network. From the application's viewpoint, TCP transfers a contiguous stream of bytes. TCP does this by grouping the bytes in TCP segments, which are passed to the underlying network protocol for transmission to the destination. TCP itself decides how to segment the data, and it may forward the data at its own convenience. If the application on one end writes 10 bytes, followed by a write of 20 bytes, followed by 50 bytes, the application at the other end cannot recognize what size the individual writes were.

TCP connections are full duplex. Once a TCP connection is established, application data can flow in both directions simultaneously. TCP provides flow control; that is, it provides a means for the receiver to govern the amount of data sent by the sender. This control is achieved by returning a "window" with every ACK indicating a range of acceptable sequence numbers beyond the last segment successfully received. The window indicates an allowed number of octets that the sender may transmit before receiving further permission. Earlier TCP implementations started a connection with the sender injecting multiple segments into the network, up to the window size advertised by the receiver. While this approach is acceptable when the two hosts are on the same LAN, a problem can arise if there are routers and slower links between the sender and the receiver. Some intermediate router must queue the packets, and that router can run out of space. This problem is often called congestion. Jacobson [10] shows how this naive approach can reduce the throughput of a TCP connection drastically. Slow start, congestion avoidance, fast retransmit, and fast recovery are the principal algorithms used to deal with congestion. These algorithms require that two variables be maintained for each connection: congestion window (cwnd) and slow start threshold size (ssthresh). The sender can transmit up to the minimum of the congestion window and the advertised window. The congestion window is flow control imposed by the sender, while the advertised window is flow control imposed by the receiver. The former is based on the sender's assessment of perceived network congestion, and the latter is related to the amount of available buffer space at the receiver for this connection.

TCP's congestion control algorithm operates as follows:

- Cwnd is initialized to one segment and ssthresh to a very large value.
- If cwnd is less than or equal to ssthresh, TCP performs slow start; otherwise TCP performs congestion avoidance.
- When congestion is indicated by a timeout, half of the current window size (the minimum of cwnd and the receiver's advertised window, but at least two segments) is saved in ssthresh, and cwnd is set to one segment (to enter slow start).
- When congestion is indicated by the reception of three consecutive duplicate ACKs, fast retransmit, fast recovery, and congestion avoidance are performed.

With slow start, cwnd begins at one segment and is incremented by one segment every time an ACK is received. This opens the window exponentially: send one segment, then two, then four, and so on.

With congestion avoidance, cwnd is incremented by $(\text{segsize} * \text{segsize}) / \text{cwnd}$ each time an ACK is received, where "segsize" is the size of segment in bytes (cwnd is maintained in bytes). This is a linear growth of cwnd, compared to slow start's exponential growth. The increase in cwnd is at most one segment each round-trip time (regardless of how many ACKs are received in that round-trip time).

Fast retransmit assumes that three or more duplicate ACKs received in a row strongly indicate that a segment has been lost. It retransmits the apparently missing segment without waiting for the retransmission timer to expire. It sets ssthresh to one-half of the current window size (the minimum of cwnd and the receiver's advertised window, but at least two segments). It also sets cwnd to ssthresh plus three times the segment size (this inflates the congestion window by the number of segments that have left the network and that the other end has cached) and enters fast recovery. (The reason for not performing slow start in this case is that the receipt of the duplicate ACKs tells TCP more than just a packet has been lost. Since the receiver can generate the duplicate ACK only when another segment is received, that segment has left the network and is in the receiver's buffer. That is, data is still flowing between the two ends, and TCP does not want to reduce the flow abruptly by going into slow start)

Fast recovery increments cwnd by the segment size each time a duplicate ACK arrives (thereby inflating the congestion window for the additional segment that has left the network) and transmits a packet, if allowed. When the next ACK arrives that acknowledges the retransmitted data, it sets cwnd to ssthresh and enters congestion avoidance.

A widely known implementation that includes the above-mentioned congestion control algorithm is known as TCP Reno. Figure 1 illustrates how TCP Reno works in the face of a single dropped segment in a window of data. Let us assume that the advertised window is fairly large throughout this example. The first unacknowledged segment (U) is 4, and the current window size (W) is 6 segments (cwnd is 6 segments). Hence, the sender transmits segments 4 through 9. Segment 4 gets lost in transit. The receiver sends duplicate ACKs for segment 4 when it receives the subsequent segments. After receiving 3 consecutive duplicate ACKs for segment 4, the sender enters fast retransmit. It sets ssthresh to 3 ($W/2$), retransmits segment

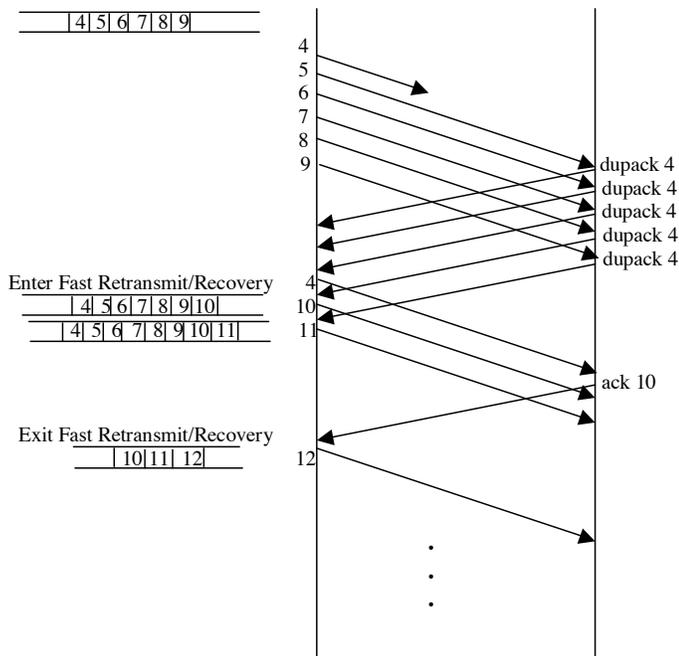


Figure 1. Behavior of TCP Reno in the presence of a single dropped segment in a window of data

4, sets $cwnd$ (and thus W) to 6 ($W/2 + 3$), and enters fast recovery. Since 6 unacknowledged segments are already outstanding, it cannot send any new data. When the sender receives another duplicate ACK for segment 4 (triggered by segment 8), it increases $cwnd$ to 7 segments and sends segment 10. When the sender receives another duplicate ACK for segment 4 (triggered by segment 9), it increases $cwnd$ to 8 segments and sends segment 11. When the receiver gets the retransmitted segment 4, it sends ACK for segment 10 (this is represented as “ack 10” in the figure. Actually, this will be the sequence number of the first octet in segment 10, and it indicates that the receiver has received up to and including segment 9). When the sender receives ACK for segment 10, it sets the $cwnd$ to $ssthresh$, which is 3, and exits fast recovery to enter congestion avoidance. The sender window allows the sender to transmit segments 10, 11, and 12. As it has already sent segments 10 and 11, it sends segment 12.

TCP Reno makes an optimistic assumption that only one segment in the window is dropped. When multiple segments are dropped from one window of data, the sender often has to wait for a retransmit timer before recovering. Figure 2 illustrates what happens when multiple segments are dropped from one window of data. When the sender receives ACK for segment 8 (represented as “ack 8” in the figure), it sets $cwnd$ to 3 ($ssthresh$) and exits fast recovery to enter congestion avoidance. The sender does not get enough duplicate ACKs to enter fast retransmit, and thus the sender has to wait for the retransmission timer to expire (even if it gets enough duplicate ACKs to enter fast

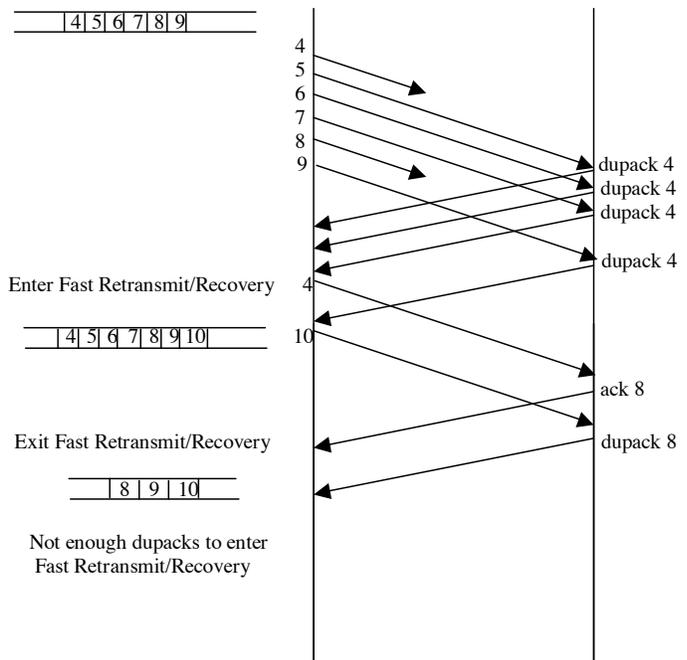


Figure 2. Behavior of TCP Reno in the presence of multiple dropped segments in a window of data

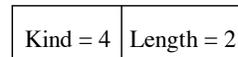


Figure 3. SACK-permitted option

retransmit again, it would unnecessarily reduce the congestion window again).

Modifications have been proposed to the fast recovery algorithm to handle multiple losses in a window of data. When a partial ACK is received (the ACKs that are less than $U+W$, where U and W are the first unacknowledged segment and the window size, respectively, when fast retransmit is entered), it retransmits the current “first unacknowledged segment” in the window and remains in fast recovery. In this way, when multiple segments are lost from a single window of data, the modified implementation, widely known as TCP New-Reno, can recover without a timeout, retransmitting one lost segment per round-trip time until all of the lost segments from that window have been retransmitted. It remains in fast recovery until all of the data outstanding when fast recovery was initiated has been acknowledged.

3 Selective Acknowledgment Scheme

Even with the New-Reno optimization, TCP ends up retransmitting at most one dropped segment per round-trip time. In order to improve this process further, the TCP selective acknowledgment (SACK) mechanism was defined in RFC 2018 [14] and later extended in RFC 2883 [7]. SACK helps TCP recover faster by providing additional

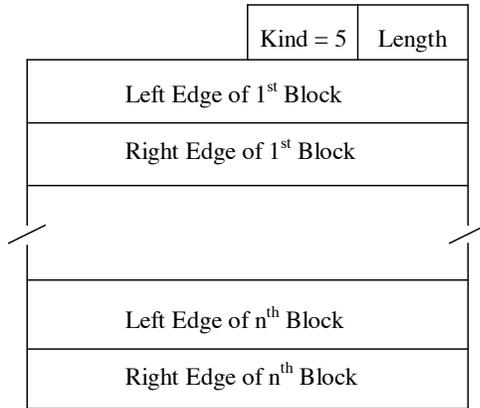


Figure 4. SACK option

information about the state of congestion. The selective acknowledgment extension uses two new TCP options. The first one is an enabling option, SACK-permitted option, that may be sent in a SYN segment to indicate that the SACK option may be used once the connection is established. The other one is the SACK option itself, which may be sent over an established connection once permission has been given by the SACK-permitted option. The SACK-permitted option is shown in Figure 3 and the SACK option in Figure 4.

SACK options will be included in all ACKs that do not acknowledge the highest sequence number in the data receiver’s queue. In this situation the network has lost or misordered data, so that the receiver holds noncontiguous blocks of data in its queue. Each noncontiguous block of data queued at the data receiver is defined in the SACK option by two 32-bit unsigned integers. “Left Edge of a Block” is the first sequence number of this block; “Right Edge of a Block” is the sequence number immediately following the last sequence number of this block. The SACK option does not change the meaning of the “Acknowledgement Number” field. A SACK option that specifies “n” noncontiguous blocks will have a length of “8*n+2” octets, so the 40 bytes available for TCP options would allow TCP to specify a maximum of 4 blocks. Of course, if other TCP options are introduced, they will compete for the 40 bytes, and the limit of 4 may be reduced further. We note that the receiver is permitted to discard data in its queue that has not been acknowledged to the data sender, even if the data has already been reported in a SACK option. This situation might happen if the receiver runs out of buffer space; hence, the sender will not discard the data reported in the SACK option until it gets an ACK for that data.

RFC 2018 does not address the use of the SACK option when acknowledging a duplicate segment; RFC 2883, on the other hand, does specify the use of SACK blocks when the SACK option is used in reporting a duplicate segment. The term D-SACK (duplicate SACK) is used to refer to a SACK block that reports a duplicate segment. The D-SACK block provides information about the duplicate segment that triggered this ACK. If present, it will be the first

block in the SACK option. A D-SACK block is used only to report a duplicate contiguous sequence of data received by the receiver in the most recent packet. Each duplicate contiguous sequence of data received is reported in at most one D-SACK block (the receiver sends two identical D-SACK blocks in subsequent packets only if the receiver receives two duplicate segments). If the D-SACK block reports a duplicate contiguous sequence from a (possibly larger) block of data in the receiver’s data queue above the cumulative acknowledgment, then the second SACK block in that SACK option should specify that (possibly larger) block of data.

Several studies have been conducted on the performance of TCP SACK. TCP SACK was shown to perform better than TCP Tahoe and TCP Reno in [3]. Charalambous et al. [2] show that, even though TCP SACK is more efficient than TCP Reno and TCP New-Reno, the throughput obtained by TCP SACK is much less than optimal. In [4], Floyd addresses various issues related to the performance of TCP SACK. The author expresses concern about the limit on the number of SACK blocks that can be carried by an acknowledgment packet. As SACK is usually implemented along with the TCP timestamp option, an acknowledgment packet can carry a maximum of only 3 SACK blocks. If D-SACK is used, the first SACK block will be used to carry information about the duplicate segment that triggered the acknowledgment. This further reduces the amount of actual SACK information that can be carried in the ACK packets. This maximum number decreases further in the presence of other TCP options. As more and more options to TCP are included, especially in the wireless environment, the issue with the low limit on the maximum number of SACK blocks needs to be addressed. Figure 5 shows an example where the TCP (with SACK) sender unnecessarily retransmits packets because of the above-mentioned limitation. Assume the left edge of the sender’s window is 3500 and that the sender transmits a burst of 12 segments, each containing 500 bytes of data. The second, seventh, ninth, and eleventh (last) segments are dropped. The receiver acknowledges the first segment normally. The third, fourth, fifth, sixth, eighth, tenth, and twelfth segments trigger SACK options. The fifth, sixth, and seventh ACK packets are dropped. The SACK option in the eighth ACK packet does not have enough space to report that the sixth segment has already been received, and thus the sender unnecessarily retransmits that segment. The example assumes that there is room for exactly 3 SACK blocks in the ACK packets.

4 Improved Selective Acknowledgment Scheme

We propose an alternative way to convey information about the noncontiguous blocks of data that have been received successfully. We call this new scheme improved selective acknowledgment (ISACK). For each block, instead of sending the absolute value of the left and the right edge, ISACK sends the offset of the left edge from the 32-bit “(cumulative) Acknowledgment Number” field in the TCP

Triggering Segment	ACK	1 st Block		2 nd Block		3 rd Block	
		Left Edge	Right Edge	Left Edge	Right Edge	Left Edge	Right Edge
3500	4000						
4000 (lost)							
4500	4000	4500	5000				
5000	4000	4500	5500				
5500	4000	4500	6000				
6000	4000 (lost)	4500	6500				
6500 (lost)							
7000	4000 (lost)	7000	7500	4500	6500		
7500 (lost)							
8000	4000 (lost)	8000	8500	7000	7500	4500	6500
8500 (lost)							
9000	4000	9000	9500	8000	8500	7000	7500

← Sender would retransmit segment 4000

← Sender would retransmit segment 6000 (unnecessary)

Figure 5. Limitation with TCP SACK

header and the block size. Similar to the SACK extension, ISACK uses 2 new TCP options. The first one is an enabling option, ISACK-permitted option, that may be sent in a SYN segment to indicate that the ISACK option may be used once the connection is established. The other one is the ISACK option itself, which may be sent over an established connection once permission has been given by the ISACK-permitted option. The ISACK-permitted option is shown in Figure 6, and the ISACK option in Figure 7. For a given packet, the number of bits used to represent the offsets is given by $\text{ceil}(\log_2(\text{maxoffset}))$, where maxoffset is the largest among the offsets and the number of bits used to represent the size of the blocks is given by $\text{ceil}(\log_2(\text{maxsize}))$, where maxsize represents the size of the largest block. Two 1-byte fields “Offset” and “Size” (in addition to the “Kind” and “Length” fields) are used to indicate the number of bits used to represent “Offset for Left Edge of a Block” and “Size of a Block,” respectively.

Kind = 27	Length = 2
-----------	------------

Figure 6. ISACK-permitted option

Except for this change in the representation and interpretation of the start and end sequence of the noncontiguous blocks of data that have been received successfully, the behavior of ISACK is same as that of SACK. For example, the first ISACK block specifies the noncontiguous block of data containing the segment that triggered this ACK, unless that segment advanced the “Acknowledgment Number” field in the header. This ensures that the ACK with the ISACK option reflects the most recent change in the data receiver’s buffer queue. An example will help clarify how ISACK works and how it differs from SACK. Assume that the left window edge is 5000 and that the sender transmits a burst of 11 segments, each containing 500 bytes of data. The second, fourth, sixth, eighth, and tenth (last) segments

Kind = 29	Length	Offset	Size
Offset for Left Edge of 1 st Block			
Size of 1 st Block			
//			
Offset for Left Edge of n th Block			
Size of n th Block			

Figure 7. ISACK option

are dropped. The receiver ACKs the first packet normally. The third, fifth, seventh, ninth, and eleventh segments trigger SACK / ISACK options. Table 1 illustrates how SACK handles the scenario, and Table 2 shows how ISACK handles it.

We see that, with SACK, by the ninth segment (sequence number 9000), 34 (4*8+2) bytes (of the 40 bytes available for the TCP options) would be used up by sending information about 4 blocks. When the eleventh segment is received, there is not enough space to send information about all 5 blocks. The information about the third segment (octets 6000 – 6500) cannot be sent.

With the ISACK scheme, on the other hand, when the eleventh segment is received, only 18 bytes are needed to convey information about all 5 noncontiguous blocks. In this case,

maxoffset = 4500
 So, the number of bits used to represent “Offset for Left Edge of a Block” = $\text{ceil}(\log_2(4500)) = 13$
 maxsize = 500

Table 1. Behavior of SACK

Triggering Segment	ACK	1 st Block		2 nd Block		3 rd Block		4 th Block	
		Left Edge	Right Edge						
5000	5500								
5500 (lost)									
6000	5500	6000	6500						
6500 (lost)									
7000	5500	7000	7500	6000	6500				
7500 (lost)									
8000	5500	8000	8500	7000	7500	6000	6500		
8500 (lost)									
9000	5500	9000	9500	8000	8500	7000	7500	6000	6500
9500 (lost)									
10000	5500	10000	10500	9000	9500	8000	8500	7000	7500

Table 2. Behavior of ISACK

Triggering Segment	ACK	1 st Block		2 nd Block		3 rd Block		4 th Block		5 th Block	
		Offset	Size								
5000	5500										
5500 (lost)											
6000	5500	500	500								
6500 (lost)											
7000	5500	1500	500	500	500						
7500 (lost)											
8000	5500	2500	500	1500	500	500	500				
8500 (lost)											
9000	5500	3500	500	2500	500	1500	500	500	500		
9500 (lost)											
10000	5500	4500	500	3500	500	2500	500	1500	500	500	500

So, the number of bits used to represent “Size of a Block” = $ceil(\log_2(500)) = 9$

The total number of bits required by the ISACK option is 8 (Kind) + 8 (Length) + 8 (Offset) + 8 (Size) + 5*13 (offsets) + 5*9 (sizes) = 142 bits (18 bytes)

The improvement is more pronounced when TCP has to use the timestamp option with SACK; then SACK can convey information about only three blocks, whereas ISACK can convey information about all the 5 blocks and still 12 bytes would remain unused in the available TCP option space.

Duplicate ISACK

We use the term D-ISACK to refer to an ISACK block that reports a duplicate segment. A D-ISACK block (if present) would be the first block in the ISACK option. The start and end sequence of a D-ISACK block may be less than the Acknowledgement Number, whereas the start and end sequence of an ISACK block is always be greater than the Acknowledgment Number. Hence, ISACK needs a simple modification to handle the D-ISACK extension. Since the start sequence of a D-ISACK block can be less than the Acknowledgment Number, there should be a way to indicate to the sender whether to add or subtract the Offset for Left Edge of 1st Block from the Acknowledgment Number to get the start sequence of the D-ISACK (first) block. Three bits each in the Offset and Size fields of the

ISACK option remain unused (Offset for Left Edge of a Block and Size of a Block can never be greater than 2^{32} and thus Offset and Size can never be more than 32). We use the most significant bit in the Offset field to indicate whether to add (if that bit is 0) or subtract (if that bit is 1) the Offset for Left Edge of 1st Block to or from the Acknowledgment Number for the D-ISACK (first) block. For the subsequent (ISACK) blocks, the Offset for Left Edge of a Block should always be added to the Acknowledgment Number to get the start sequence of a block. The number of bits used to represent the Offset for Left Edge of a Block is given by “Offset & 0x1f.”

5 Adaptive Selective Acknowledgment Scheme

Kind = 28	Length = 2
-----------	------------

Figure 8. ASACK-permitted option

ISACK imposes a little more processing overhead than does SACK. Hence, it is desired that ISACK be used only when SACK is not able to convey information about all the noncontiguous blocks of data that have been received successfully and queued. We propose an adaptive selective acknowledgment (ASACK) scheme that dynamically switches between SACK and ISACK. ASACK uses SACK

as long as it can convey all the information with the available TCP option space. If SACK cannot convey all the information, ASACK switches to ISACK. The introduction of the ASACK mechanism necessitates a new enabling TCP option, the ASACK-permitted option, that may be sent in a SYN segment to indicate that either the SACK or ISACK option may be used once the connection is established. The ASACK-permitted option is shown in Figure 8.

6 Conclusion

We presented an improved selective acknowledgment scheme (ISACK) that addresses the limitations with the current selective acknowledgment mechanism. We showed how the proposed scheme can handle the duplicate SACK extension proposed for SACK. We used an example to demonstrate how the proposed ISACK scheme works. As ISACK can convey more information than SACK, it is more resilient to the high packet error rates often seen in wireless environments. We also developed an adaptive scheme (ASACK) that makes use of the advantages of both SACK and ISACK.

Acknowledgment

This work was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract W-31-109-ENG-38.

References

- [1] M. Allman, V. Paxson, and W. Stevens. RFC 2581: TCP Congestion Control, 1999.
- [2] C. Charalambous, V. Frost, and J. Evans. Performance of TCP Extensions on Noisy High BDP Networks. *IEEE Communications Letters*, 3(10):294–299, 1999.
- [3] K. Fall and S. Floyd. Simulation-Based Comparisons of Tahoe, Reno and SACK TCP. *ACM SIGCOMM Computer Communication Review*, 26(3):5–21, 1996.
- [4] S. Floyd. Issues of TCP with SACK. Technical report, Lawrence Berkeley National Laboratory, January 1996.
- [5] S. Floyd. RFC 3649 (Experimental): HighSpeed TCP for Large Congestion Windows, 2003.
- [6] S. Floyd and T. Henderson. RFC 2582: The NewReno Modification to TCP’s Fast Recovery Algorithm, 1999.
- [7] S. Floyd, J. Mahdavi, M. Mathis, and M. Podolsky. RFC 2883: An Extension to the Selective Acknowledgment (SACK) Option for TCP, 2000.
- [8] Y. Gu and R. Grossman. UDT (UDP based Data Transfer Protocol): An Application Level Transport Protocol for Grid Computing. *Presented at the Second International Workshop on Protocols for Fast Long-Distance Networks*, 2004.
- [9] E. He, J. Leigh, O. Yu, and T. DeFanti. Reliable Blast UDP: Predictable High Performance Bulk Data Transfer. In *Proceedings of the IEEE International Conference on Cluster Computing*, pages 317–324. IEEE Computer Society, 2002.
- [10] V. Jacobson. Congestion Avoidance and Control. *ACM SIGCOMM Computer Communication Review*, 18(4):314–329, 1988.
- [11] V. Jacobson, R. Braden, and D. Borman. RFC 1323: TCP Extensions for High Performance, 1992.
- [12] D. Katabi, M. Handley, and C. Rohrs. Congestion Control for High Bandwidth-Delay Product Networks. *ACM SIGCOMM Computer Communication Review*, 31(4):89–102, 2002.
- [13] T. Kelly. Scalable TCP: Improving Performance in Highspeed Wide Area Networks. *ACM SIGCOMM Computer Communication Review*, 33(2):83–91, 2003.
- [14] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow. RFC 2018: TCP Selective Acknowledgment Options, 1996.
- [15] J. Postel. RFC 793: Transmission Control Protocol, 1981.

Manuscript has been created by the University of Chicago as Operator of Argonne National Laboratory (“Argonne”) under Contract No. W-31-109-ENG-38 with the U.S. Department of Energy. The U.S. Government retains for itself, and others acting on its behalf, a paid-up, nonexclusive, irrevocable worldwide license in said article to reproduce, prepare derivative works, distribute copies to the public, and perform publicly and display publicly, by or on behalf of the Government.