Formal Specification and Verification of Safety and Performance of TCP Selective Acknowledgment

Mark A. Smith and K. K. Ramakrishnan, *Member, IEEE*

***Abstract*—We present a formal specification of the selective acknowledgment (SACK) mechanism that is being proposed as a new standard option for TCP. The formal specification allows one to reason about the SACK protocol; thus, we are able to formally prove that the SACK mechanism does not violate the safety properties (reliable, at most once, and in order message delivery) of the acknowledgment (ACK) mechanism that is currently used with TCP. The new mechanism is being proposed to improve the performance of TCP when multiple packets are lost from one window of data. The proposed mechanism for implementing the SACK option for TCP is sufficiently complicated that it is not obvious that it is indeed safe, so we think it is important to formally verify its safety properties.**

In addition to safety, we are also able to show that SACK can improve the time it takes for the sender to recover from multiple packet losses. With the additional information available at a SACK sender, the round-trip time that a cumulative ACK sender waits before retransmitting each subsequent packet lost after the very first loss can be saved. We also show that SACK can improve performance even with window sizes as small as four packets and in situations where acknowledgment packets are lost.

***Index Terms*—Congestion control, formal verification, I/O automata, TCP performance, TCP SACK.**

I. Introduction

TRANSMISSION Control Protocol (TCP) offers applications the semantics of a reliable, flow-controlled channel. The acknowledgment (ACK) mechanism of TCP is an important part of what makes the protocol reliable. By reliable, we mean data from the sender is not lost, duplicated, or received out of order. TCP guarantees these properties, which we refer to as safety properties, even though the underlying communication medium may lose, duplicate, or reorder packets. We assume corrupted packets are dropped.

Selective Acknowledgments (SACK) [10] have been proposed as a complement to the traditional approach of using cumulative acknowledgments for TCP. SACK is proposed as a standard option to be used by cooperating senders and receivers. The receiver can take advantage of the SACK option to report that it has received multiple packets out of sequence. A sender receiving a SACK has the opportunity to retransmit the packets that comprise the holes in the sequence number space as indicated in the SACK. The new functionality that SACK introduces is the potential for earlier recovery, especially when multiple packets are lost in round-trip time (RTT). This quicker recovery may also result in higher throughput because the more severe congestion recovery mechanisms are not invoked.

The SACK mechanism includes sufficient additional complexity that we believe it is important to examine whether it operates correctly. It is not obvious from reading the English language specification of [10] that it satisfies the safety properties of the ACK mechanism. Simulation experiments have been done to understand the performance improvement with SACK [1], and while these lend confidence that the protocol operates as expected, simulations do not ensure that the protocol is correct. Formal specifications and verification help significantly in ensuring that protocols operate correctly. Therefore, we feel that a formal specification and verification of the safety properties of the mechanism is useful. We have developed a formal specification of the SACK mechanism using the I/O automaton model of Lynch and Tuttle [7]. The formal specification of the SACK mechanism allows one to reason about the protocol in a rigorous manner. For the formal verification of safety, we use invariant assertion and simulation (refinement) techniques. These methods are used for proving trace inclusion relationships between concurrent systems. Trace inclusion means the external behaviors of one system is the subset of the external behaviors of another system. For this verification, we use the methods to show that the external behaviors of TCP with the SACK option, which we refer to simply as SACK, is a subset of the external behaviors of a simple abstract specification for end-to-end reliable message delivery. We use the formalization of simulations developed by Lynch and Vaandrager [8].

A key aspect of our formal specification of the SACK mechanisms is that it allows more nondeterminism than the English language specification [10]. Our specification focuses on key aspects of the protocol needed for safety while leaving certain aspects of the protocol, such as retransmission strategy, unspecified. This means our correctness proof is quite general and holds for any implementation of the protocol that uses a specific retransmission strategy or other specific behaviors that we leave nondeterministic in our specification.

We extend the specification used to prove safety properties to include time using the general timed automaton model of Lynch [6]. We use this model to calculate the latency of packets in a window of data in a worst-case scenario and prove that SACK can lead to improved performance relative to the cumulative ACK mechanism. In fact, we prove that in situations where the multiple packet loss comes early in the transmission of a window of data, the improved performance with SACK is proportional to $\mathrm{RTT} * (k-1)$, where k is the number of packets

Manuscript received May 12, 1999; revised March 7, 2001; approved by IEEE/ACM Transactions on Networking Editor K. Calvert.

M. A. Smith is with Bell Labs, Murray Hill, NJ 07974 USA (e-mail: massmith@lucent.com).

K. K. Ramakrishnan is with TeraOptic Networks, Inc., Sunnyvale, CA 94085 USA (e-mail: kk@teraoptic.com).

Publisher Item Identifier S 1063-6692(02)03106-0.

lost in a window. We also show that SACK can improve performance even when window sizes are small and/or when acknowledgment packets are lost.

In the next section, we present an informal description of both the cumulative ACK and SACK mechanisms. In Section III, we present the abstract requirements of the reliable message delivery service, and we also present a brief description of the formal model we use in the paper. In Section IV, we present the formal specification of the SACK mechanisms. Section V has the proof of safety along with a short description of the proof techniques we use. In Section VI, we prove that SACK can lead to improved performance, and we conclude in Section VII. The paper also contains two appendices. Appendix I gives formal definitions for the notation used in the abstract specification of the reliable message delivery problem and the formal description of the SACK mechanism, and Appendix II contains the proof of the invariants in Section V.

II. Informal Description of the Acknowledgment Mechanisms

TCP uses a *sliding window* mechanism for its flow control, acknowledgment, and retransmission policy. The basic idea is that there is a window of size $n \geq 0$, that determines how many successive segments of data can be sent in the absence of a new acknowledgment. Each segment of data is sequentially numbered, so the sender is not allowed to send segment $i+n$ before segment i has been acknowledged. Thus, if i is the largest acknowledgment number received by the sender, there is a window of data numbered i to $i+n-1$ which the sender can transmit. As successively higher numbered acknowledgments are received, the window slides forward. The acknowledgment mechanism is cumulative in that if the receiver acknowledges segment k, it means it has successfully received all segments up to and including k. Segment k is acknowledged by sending a request for segment $k+1$. Data that is transmitted is kept on a retransmission buffer until it has been acknowledged. In a simple go-back-n protocol, if $k < n+i$, the sender may retransmit segments $k+1$ to $n+i$ from the retransmission buffer. However, in TCP the decision to retransmit these segments depends on the receipt of duplicate acknowledgments and on timeouts. With TCP Reno, only the first packet in the retransmission buffer is sent. Subsequently, the sender waits until the retransmitted packet is acknowledged. This strategy potentially reduces the unnecessary retransmissions compared to the simple go-back-n protocol.

Of particular interest are the mechanisms which TCP uses to recover from loss, including algorithms for *fast retransmit* [4]. With fast retransmit, when the source receives d duplicate acknowledgments (e.g., $d = 3$) for the same segment (say, k), it determines that segment k was lost. The source chooses to retransmit segment k right away, rather than wait for a retransmission timer to expire. It must be noted that the sender retransmits one packet (or segment for the purposes of this paper) only.

The limitation of the cumulative acknowledgment strategy is that it can only indicate that every segment up to k has been received and $k+1$ has not been received. When multiple packets are lost from a window, the throughput of TCP can suffer greatly. After retransmitting the first packet in the retransmission buffer, the sender may be forced to wait for a retransmission timeout to retransmit subsequent "holes" in the receiver's sequence number space.

To remedy the problem that occurs when multiple packets in a round trip are lost, a selective acknowledgment mechanism is being proposed as a new standard option for TCP [10]. The mechanism allows the receiver of data to acknowledge noncontiguous and isolated blocks of data that have been received and queued, in addition to the cumulative acknowledgment of contiguous data. By isolated, we mean the segment just below the block and just above the block have not been received. Each block is defined by a pair of sequence numbers. The first number is the left edge of the block and is the sequence number of the first segment of data in the block that was received. The second number is the right edge of the block and is the number immediately following the last sequence number of the block that was received. The retransmission strategy of the sender also changes to use the additional information available with selective acknowledgment. Now, in addition to the data segments in the retransmission buffer, there is a flag bit which indicates whether a segment has been "SACKed." A segment with the SACKed bit turned on is not retransmitted, but segments with the SACKed bit turned off and sequence number less than the highest SACKed segment are available for retransmission.

A. An Example With Cumulative ACK

In Fig. 1, we show a simple example that illustrates how the (cumulative) ACK mechanism works with TCP Reno [5]. The figure shows the retransmission buffer of the sender and the buffer at the receiver. Let the window size be 8 for this example. The threshold of the number of duplicate acknowledgments that need to be received before the fast retransmit algorithm is triggered is assumed to be 3. The numbers in the buffers and the numbers on the segments sent by the sender represents the actual segment of data and is the sequence number of that segment of data. The variable `snd_una` is the sequence number of the segment at the head of the retransmission buffer. It is also the oldest unacknowledged segment of data. The `rcv_nxt` variable is the next contiguous segment of data expected by the receiver. This variable is the acknowledgment number that the receiver sends back to the sender. The execution illustrated in Fig. 1 begins with the sender transmitting segment 26. Next, segment 27 gets transmitted, but is lost due to, say, congestion. Subsequently, segments 28 and 29 are delivered. The acknowledgments generated by the receiver on receipt of segments 26, 28, and 29 all indicate that the next segment expected is segment 27. In the example, segment 30 is also lost. Thus, two segments 27 and 30 are lost in the current window of 8 segments. Subsequently, when segment 31 is received, the acknowledgment generated triggers the fast retransmit algorithm. This causes segment 27 to be retransmitted without waiting for a retransmit timeout. Notice that even after the source retransmits segment 27, acknowledgments are received with the next expected segment being 27 (sent in response to packets 32 and 33 sent before the retransmission of segment 27). This allows the sender to send new segments, but not retransmit any more segments from the retransmission buffer, since it does not know which segments need to be sent.

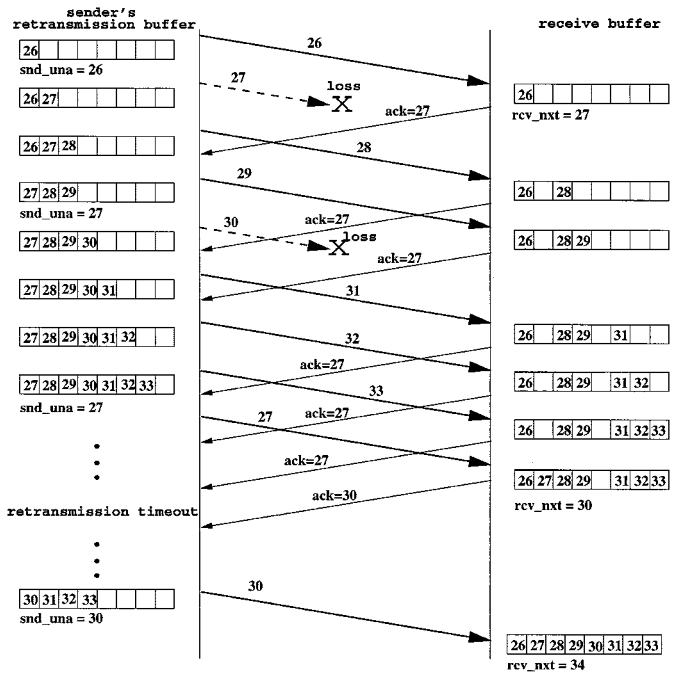


Fig. 1. Example illustrating the workings of the ACK mechanism of TCP.

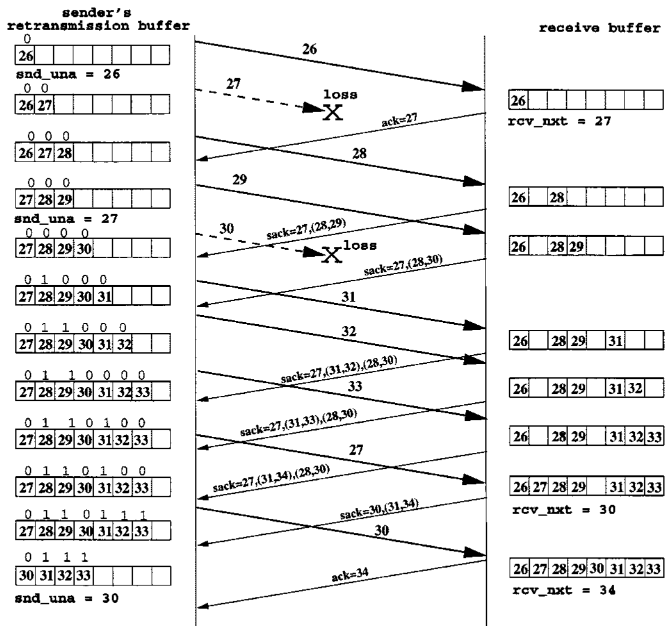


Fig. 2. Example illustrating the workings of the SACK mechanism of TCP.

If it were to decide to retransmit, it would have to retransmit the next segment in the buffer, which is segment 28. But this would be a wasteful retransmission. Hence, the sender desists from retransmitting any new packets (but can use the opportunity to send new segments if the window allows it). It is only after a new acknowledgment is received indicating successful receipt of segment 27 can the sender potentially consider retransmitting another packet from its retransmission buffer. However, the second loss in a window is interpreted as a more serious situation—hence the fast retransmission algorithm is not invoked to recover from this loss. The second segment that was lost in this window (30) is not retransmitted until a retransmission timeout. Because this timeout is necessarily large, the sender's window is likely to be shut. Thus, during this timeout, the sender is unable to make progress, resulting in degraded throughput. This timeout also results in the sender dropping the congestion window size to 1 based on the congestion control algorithms described in [4].

B. An Example With SACK

Fig. 2 illustrates how the SACK mechanism works on the same set of data as in Fig. 1. With selective acknowledgment, the receiver sends back the regular cumulative acknowledgment number and SACK blocks. In the proposed implementation of the SACK mechanism described in [10], there are at most three SACK blocks in an acknowledgment packet.

Initially, when the receiver gets segment 28 (after the loss of segment 27), it sends a SACK for segment 27 and a further SACK block indicating that segment 28 was received and segment 29 was awaited after that. When segment 31 is received after the loss of segment 30, the SACK sent indicates that 27 (the portion representing the cumulative ACK) is awaited, and furthermore that two blocks of data were received with an intervening gap of segment 30. The regular fast retransmit algorithm is triggered on receipt of third duplicate SACK indicating that segment 27 was lost. The source retransmits segment 27. But at the same time, the sender has the information that segment 30 has also been lost as indicated in the SACK. Since the source now has the information of specifically which segments have been lost (segment 27 and 30), it has the ability to retransmit more than one of the lost segments. Therefore, the sender can also retransmit segment 30 and fill the second hole in the sequence number space. The sender does not need to wait for a retransmit timeout for retransmitting segment 30.

A further detail to be observed is the maintenance of the SACK flags at the source, associated with the segments still in the retransmission buffer. In the figure SACKed segments are not removed from the retransmission buffer even though in this example, these segments are not retransmitted. However, the SACK mechanism allows the receiver to drop segments that have been SACKed if it runs out of buffer space. Thus, it is possible that these segments may need to be retransmitted. Since they are not retransmitted if the SACK flag is set, there must be a mechanism for resetting the flags to 0. In the SACK mechanism proposed for TCP when a retransmission timeout expires for any sequence of data, all the SACKed bits in the retransmission buffer are reset to 0.

Thus, the SACK option allows the sender to recover from losing multiple packets in a round-trip time, and "fill" all the "holes" in the receiver's sequence number space based on the SACK blocks received. Further, it allows for the separation of the flow control and congestion control mechanisms from being intricately tied to the error recovery procedures, as we illustrated in the example. It allows the source the ability to be somewhat more aggressive in both retransmitting from the buffer on multiple packet losses, and not dropping the congestion window down all the way to 1.

III. Formal Model and Service Requirements

The safety properties we want to show for TCP with the SACK mechanism are that data from the sender is not lost, duplicated, or received out of order. Our model does not capture data corruption, and we assume corrupted packets are discarded. In this work, we do not try to verify the safety of TCP in its entirety. We are only concerned with the effects of replacing the ACK mechanism with the SACK mechanism. Therefore, we assume that connection setup and teardown work correctly, and that crash recovery works correctly. Consequently, in our modeling of the SACK mechanism, we assume that the connection between the sender and receiver is already established and that there are no crashes. Before we present the specification, we give a brief description of the formal model we use.

A. Automaton Model

The formal model we use to describe the acknowledgment mechanisms is based on the I/O automaton model of [7]. An automaton A consists of four components: 1) a set $\mathrm{states}(A)$ of states; 2) a nonempty set $\mathrm{start}(A)$ $\subseteq$of $\mathrm{states}(A)$ of start states; 3) a set $\mathrm{acts}(A)$ of actions; and 4) a set $\mathrm{steps}(A) \subseteq \mathrm{states}(A) \times \mathrm{acts}(A) \times \mathrm{states}(A)$ of steps. The set $\mathrm{acts}(A)$ can be partitioned into three disjoint sets, $\mathrm{in}(A), \mathrm{out}(A),$, and $\mathrm{int}(A)$ of *input actions*, *output actions*, and *internal actions*, respectively. The union of the input actions and output actions we denote as *external actions*, those actions visible to the environment.

When an automaton runs, it generates a string representing an execution of the system it models. An *execution fragment* α of automaton A is a finite or infinite sequence, $s_0, a_1, s_1, a_2, \ldots$, of alternating states and actions of A starting in a start state, and if the execution fragment is finite, ending in a state such that (s_i, a_{i+1}, s_{i+1}) is a step of A for every i. We denote by $\mathrm{fstate}(\alpha)$ the first state of the execution fragment, and if it is finite $\mathrm{lstate}(\alpha)$ denotes the last state. A state s is said to be *reachable* if there exists a finite execution of A that includes s.

Suppose $\alpha = s_0, a_1, s_1, a_2, \ldots$ is an execution fragment of A. Then $\mathrm{trace}_A(\alpha)$ or $\mathrm{trace}(\alpha)$ if A is clear, is defined to be the subsequence of α consisting of only the external actions. We say that β is a trace of A if there exists an execution α of A with $\mathrm{trace}(\alpha) = \beta$.

In specifying a complex distributed system, it is useful to be able to specify each process individually and then obtain a specification of the entire system as the *parallel composition* of the specifications of the processes. The parallel composition operator $||$ in this model uses a synchronization style where automata synchronize on their common actions and evolve independently on the others.

To show that an automaton A implements another automaton B, we show a *trace* inclusion relationship between them. The set of traces of an automaton consists of the set of sequences of visible actions that the automaton can perform.

B. Service Requirements

For the formal description of the service requirements, we assume a very simple user interface—there is an input action from the user on the sender side to send data message send(m), and on the receiver side, data is passed to the user with the deliver(m) action.

```
automaton ReliableQ

signature
   input send(m: Seq[Byte])
   output deliver(m: Seq[Byte])

states
   queue: Seq[Byte] := {}

transitions
   input send(m)
   eff  queue := queue || m

   output deliver(m)
   pre  queue ≠ {}
          m ∈ prefixes(queue)
   eff  queue := tail(queue, |m|)
```

Fig. 3. Model of the requirements of the reliable message delivery problem.

We define a simple automaton which is an abstract representation of the safety properties that we want to show are satisfied by the SACK mechanism. The automaton, which we call ReliableQ is shown in Fig. 3. We describe the automaton in the style of the IOA language [2] for describing I/O automata, but we do not strictly follow the syntax of the formal language. In the IOA language, an automaton is described by first giving its name followed by the four components of the model mentioned above. That is, we have the action signature (**signature**), the states and start states (**states**), and the set of steps (**transitions**). Transitions are written in a *precondition*, *effect* fashion. That is, the states in which an action is enabled is given as a precondition, and the resulting states are given by the effects of the action.

The only state variable of the automaton is queue which has type Seq[Byte] and is initially empty. The type Seq[Byte] is an ordered list or sequence with elements of type Byte. We denote the empty sequence as $\{\}$. The input action send(m) from the user causes data m to be added to the tail of the queue. The symbol $||$ is the concatenation operator. The output action deliver(m) passes data from the head of the queue to the receiver side user. The prefixes operator returns the set of prefixes of the queue, and the tail(queue, |m|) operation removes the first $|m|$ elements from the queue. Since the queue does not lose, duplicate, or reorder data, it is easy to see that the specification ReliableQ gives the safety properties we want. The prefixes, tail and all the operators used in subsequent sections are formally defined in Appendix I.

IV. Formal Specification of SACK

TCP has the basic structure shown in Fig. 4. There is a sender, a receiver, a channel for packets from the sender to the receiver, and a channel for packets from the receiver to the sender. The protocol is only run at the sender and the receiver, but it assumes the channels, so the channels must be modeled. We model each component as an automaton, and the complete system is the parallel composition of the four component automata.

In our models of the sender and the receiver below, we specify certain state variables as unbounded integers. However, in the actual protocol, the size (2^{32}) of these variables is bounded. Certain aspects of our proof of the correctness of the protocol relies

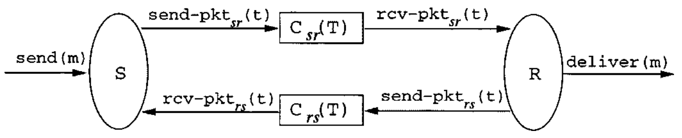


Fig. 4. Structure and the four basic components of the model for SACK.

on being able to make accurate comparisons of the relative size of some of these variables. When the variables are unbounded, it is easy to see that these comparisons are accurate. However, with a bounded number space that may wrap around, it is not so clear that these comparisons, which must now be made modulo 2^{32}, can be done accurately. TCP uses various timing mechanisms coupled with the relatively large number space to ensure that the relative size of variables are not confused. For further details and formal proofs as to why the timing mechanisms work, see [11]. The mechanisms do not vary between standard TCP and SACK TCP, so we do not focus on them here. Instead, we use unbounded counters to simplify our models and proofs.

A. Channel Automaton

The channels in our model can lose, duplicate, and reorder packets, but they do not corrupt or create spurious packets. The I/O automaton model for the channel from the sender to the receiver, $\mathtt{C}_{\mathtt{sr}}(\mathtt{T})$, is shown in Fig. 5. The model for the channel from the receiver to the sender is essentially the same, so we do not show it here. The basic difference is that the subscripts in the names of the state variable and the actions in the signatures are different and reflect the directional flow of packets in the channels. The channels have a packet type that is the channel parameter T. The state variable $\mathtt{in\_transit}_{sr}$ (for the receiver to sender channel the variable is $\mathtt{in\_transit}_{rs}$) holds the packets placed on the channel. Since the channels may have duplicate copies of a packet, this variable is a multisets of type T. The fact that it is a multiset means the packets are not ordered. Thus, packets may be received in a different order from the way they were sent.

The $\mathtt{send\text{-}pkt}_{\mathtt{sr}}(\mathtt{t})$ input action places a packet in the multiset. The complementary output action $\mathtt{rcv\text{-}pkt}_{\mathtt{sr}}(\mathtt{t})$ removes the packet from the channel. The internal action $\mathtt{drop}_{\mathtt{sr}}(\mathtt{t})$ nondeterministically removes an element from the multiset. The set minus operator for the multiset only removes one copy of an element. The internal action $\mathtt{duplicate}_{\mathtt{sr}}(\mathtt{t})$ adds one additional copy of an element to the multiset.

B. Sender Automaton

In this section, we present a formal I/O automaton model for the sender protocol of the SACK mechanism. The automaton, S, is shown in Fig. 6. We first specify the type definitions needed to describe some components of the automaton. The ByteInt type is the set of pairs that has a byte as the first element, and an integer as the second element. The type Sbyte is the set of triples formed by a byte, an integer sequence number, and a Boolean flag. The type Blk is a pair of integers, and indicates the left and right edges of a block of data.

The states of the sender includes send_buf which is a sequence of bytes, and it holds data received from the user. The retransmission buffer, retran_buf, holds data that may need

```
automaton C_sr(T: type)

signature
   input send-pkt_sr(t: T)
   internal drop_sr(t: T)
   internal duplicate_sr(t: T)
   output rcv-pkt_sr(t: T)

states
   in_transit_sr: Mset[T] := {}

transitions

   input  send-pkt_sr(t)
   eff  in_transit_sr := in_transit_sr ∪ {t}

   internal drop_sr(t)
   pre  t ∈ in_transit_sr
   eff  in_transit_sr := in_transit_sr \ {t}

   output rcv-pkt_sr(t)
   pre  t ∈ in_transit_sr
   eff  in_transit_sr := in_transit_sr \ {t}

   internal duplicate_sr(t)
   pre  t ∈ in_transit_sr
   eff  in_transit_sr := in_transit_sr ∪ {t}
```

Fig. 5. Model for the channel for packets from the sender to the receiver.

to be retransmitted. Each byte of data is grouped with its sequence number and a flag indicating whether the byte of data has been selectively acknowledged. Both buffers are initially empty. The segment variable is the current segment being sent, snd_una is the sequence number of the oldest unacknowledged byte, snd_nxt is the sequence number of the next byte to be sent, and ready_to_send is a flag that enables the sending of segments when the transmission window is open.

Input action send(m) is the action by the user that passes data to the sender, and prepare-new-seg(s) prepares a new segment to be sent. It is only enabled if the sender is not currently enabled to send a segment ($\neg$ready_to_send), the send buffer is not empty, and the available window size is greater than 0 (we assume a constant window size of WS). This action nondeterministically chooses the portion of data to be sent. This portion of data, s, must be a prefix of the send buffer and its length, written $|\mathtt{s}|$, must be less than or equal to the minimum of the maximum segment size (MSS) and the available window size. The effect of this action is to remove s from the send buffer, and then to pair each element of s with its sequence number. This pairing is done by the enum(s, snd_nxt) operation. The new list forms the segment to be sent and is assigned to the variable segment. A SACK flag that is initialized to false is added to each pair in the segment sequence before it is concatenated to the retransmission buffer. The init_flag operator performs this initialization.

The $\mathtt{send\text{-}pkt}_{\mathtt{sr}}(\mathtt{seg})$ action places a segment on the outgoing channel, $\mathtt{C}_{\mathtt{sr}}(\mathtt{T})$, of the sender, and $\mathtt{rcv\text{-}pkt}_{\mathtt{rs}}(\mathtt{ack})$ takes a simple ACK packet from $\mathtt{C}_{\mathtt{rs}}(\mathtt{T})$. First ack is checked to see that it acknowledges data that was sent, $\mathtt{snd\_una} < \mathtt{ack} \leq \mathtt{snd\_nxt}$. If this condition is true, the acknowledged data is removed from the retransmission buffer, and snd_una is updated. When a SACK packet is received, $\mathtt{rcv\text{-}pkt}_{\mathtt{rs}}(\mathtt{ack, b1, b2, b3})$, the sender sets the SACK flag of the bytes indicated by the SACK blocks to true with the assignment of retran_buf to set_sack(retran_buf, b1, b2, b3).

```
type ByteInt = tuple of Msg: Byte, Seqnum: Int
type Sbyte = tuple of Msg: Byte, Seqnum: Int, Flag: Bool
type Blk = tuple of Left: Int, Right: Int

automaton S

signature
   input send(m: Seq[Byte]), rcv-pkt_rs(ack: Int),
         rcv-pkt_rs(ack:Int, b1:Blk, b2:Blk, b3:Blk)
   internal prepare-new-seg(s: Seq[Byte])
         prepare-retran-seg(s: Seq[Sbyte]), reset-sack
   output send-pkt_sr(seg: Seq[ByteInt])

states
   send_buf: Seq[Byte] := {}
   retran_buf: Seq[Sbyte] := {}
   segment: Seq[ByteInt] := {}
   snd_una, snd_nxt: Int := 0
   ready_to_send: Bool := false

transitions

   input send(m)
   eff  send_buf := send_buf || m

   internal prepare-new-seg(s)
   pre  ¬ ready_to_send
        send_buf ≠ {}
        snd_nxt < snd_una + WS
        s ∈ prefixes(send_buf)
        1 ≤ |s| ≤ min(MSS, snd_una + WS - snd_nxt)
   eff  send_buf := tail(send_buf, |s|)
        segment := enum(s, snd_nxt)
        retran_buf := retran_buf || init_flag(segment)
        snd_nxt := snd_nxt + |s|
        ready_to_send := true

   output send-pkt_sr(seg)
   pre  ready_to_send
        seg = segment
   eff  ready_to_send := false

   input rcv-pkt_rs(ack)
   eff  if  snd_una <  ack ≤ snd_nxt then
          retran_buf := delete(retran_buf, ack)
          snd_una := ack fi

   input rcv-pkt_rs(ack, b1, b2, b3)
   eff  if snd_una <  ack ≤ snd_nxt then
          retran_buf := delete(retran_buf, ack)
          retran_buf := set_sack(retran_buf,b1,b2,b3)
          snd_una := ack fi

   internal prepare-retran-seg(s)
   pre  ¬ ready_to_send
        retran_buf ≠ {}
        s ∈  holes(retran_buf)
        1 ≤ |s| ≤ MSS
   eff  segment := remove_flag(s)
        ready_to_send := true

   internal reset-sack
   pre  true
   eff  retran_buf := unsack(retran_buf)
```

Fig. 6. SACK sender automaton.

In TCP Reno, a segment is retransmitted if the retransmission timeout expires, or a certain number of duplicate acknowledgments are received. In our model of the sender, we simplify the mechanism by allowing the sender to generate retransmission segments nondeterministically. Since the sender can nondeterministically choose between retransmitting a segment or sending a new one whenever it is enabled to send a segment, our model accommodates the TCP Reno retransmission policy and any other implementation policy for retransmission. The internal action that generates retransmission segments is `prepare-retran-seg(s)`. Since the SACK protocol does not retransmit SACKed blocks of data, the retransmission segment chosen must be from the set of unSACKed contiguous bytes. These are the known holes in the data at the receiver. The `holes` operator returns this set, and the segment retransmitted is nondeterministically chosen from the set. Again, the nondeterminism in our model allows us to accommodate any implementation policy that determines which holes to fill.

```
type Blk = tuple of Left: Int, Right: Int
type ByteInt = tuple of Msg: Byte, Seqnum: Int

automaton R

signature
   input  rcv-pkt_sr(seg: Seq[ByteInt])
   internal drop(s: Seq[ByteInt])
   output deliver(m: Seq[Byte]), send-pkt_rs(ack: Int),
         send-pkt_rs(ack:Int, b1:Blk, b2:Blk, b3:Blk)
states
   rcv_buf: Seq[ByteInt] := {}
   rcv_nxt: Int := 0
   send_ack, sack_opt: Bool := false

transitions

   input rcv-pkt_sr(seg)
   eff  send_ack := true
        if rcv_nxt ≤ last(get_seq(seg)) then
          rcv_buf := inst(rcv_buf,delete(seg,rcv_nxt))
          if seq_max_con(rcv_buf) > rcv_nxt then
            rcv_nxt := seq_max_con(rcv_buf) + 1 fi
          if rcv_nxt ≤ last(get_seq(rcv_buf)) then
            sack_opt := true
          else sack_opt := false fi
        fi

   output send-pkt_rs(ack)
   pre  send_ack ∧ ¬ sack_opt
        ack = rcv_nxt
   eff  send_ack := false

   output send-pkt_rs(ack, b1, b2, b3)
   pre  send_ack ∧ sack_opt
        ack = rcv_nxt
        b1 ∈ blocks(rcv_buf)
        b2 ∈ blocks(rcv_buf)
        b3 ∈ blocks(rcv_buf)
        |blocks(rcv_buf)| > 1 ⇒ b2 ≠ b1
        |blocks(rcv_buf)| > 2 ⇒ b3 ∉{b1,b2}
   eff  send_ack := false

   output deliver(m)
   pre  rcv_buf ≠ {}
        m ∈  cprefixes(get_data(rcv_buf))
   eff  rcv_buf := tail(rcv_buf,  |m|)

   internal drop(b)
   pre b ∈ blocks(rcv_buf)
   eff  rcv_buf := delete(rcv_buf, b.Left, b.Right)
```

Fig. 7. Automaton of the SACK receiver.

In the proposed implementation of SACK [10], whenever a retransmission timeout expires, all the SACK flags in the retransmission buffer are reset to false. In our model, we again use nondeterminism for a simpler but more general model. We allow the resetting of the flags, nondeterministically, at anytime. The internal action `reset-sack` resets the sack flags.

C. Receiver Automaton

The automaton model for the receiver protocol of SACK is shown in Fig. 7. The `rcv_buf` variable holds segments that are received until they are passed to the user. The `rcv_nxt` variable is the sequence number of the next byte of data expected by the receiver and is also the acknowledgment number sent to the sender. The variables send_ack and sack_opt are flags that enables the sending of a regular acknowledgment segment and an acknowledgment segment with SACK blocks, respectively.

The action $\texttt{rcv-pkt}_{\texttt{sr}}(\texttt{seg})$ causes the receiver to set the send_ack flag to true, which enables the sending of an acknowledgment segment. The received segment has new data if $\texttt{rcv\_nxt} \leq \texttt{last}(\texttt{get\_seq}(\texttt{seg}))$. That is, there is new data if the sequence number of the last byte of data in the segment is greater than or equal to rcv_nxt. If the segment has new data, the part of the segment that has sequence number greater than or equal to rcv_nxt, is inserted in sequence number order in the receive buffer. Informally speaking, the inst operator inserts the new segment into the receive buffer so that sequence numbers in the updated buffer remain sorted in ascending order. Elements from the new segment with the same sequence numbers as elements already in the buffer overwrite the elements in the buffer. If there is new data, rcv_nxt is updated to reflect the last contiguous piece of data that has been received. The update is done by taking the sequence number from the last element of the maximum contiguous prefix of the receive buffer, $\texttt{seq\_max\_con}(\texttt{rcv\_buf})$, and adding 1 to that number. The receiver must also determine whether to send a regular acknowledgment or a selective acknowledgment. A selective acknowledgment is sent if the receiver has noncontiguous data queued in the receive buffer. That is, if rcv_nxt does not acknowledge the highest sequence number in the receive buffer $(\texttt{rcv\_nxt} \leq \texttt{last}(\texttt{get\_seq}(\texttt{rcv\_buf})))$, then sack_opt is set to true.

If send_ack is true and sack_opt is false, then the $\texttt{send-pkt}_{\texttt{rs}}(\texttt{ack})$ action is enabled. This action sends an acknowledgment where ack is the acknowledgment number. The setting of sack_opt to true enables the $\texttt{send-pkt}_{\texttt{rs}}(\texttt{ack, b1, b2, b3})$ action, where b1, b2, and b3 are three nondeterministically chosen SACK blocks. The set of SACK blocks for the receive buffer is generated by the blocks operator. If there are at least two SACK blocks, then b1 and 2 are different. If there are at least three, then all three blocks are different. We limit the number of blocks to three in our model because the restriction in the header size of a TCP segment means that for most cases this will be maximum number of SACK blocks that can be included in an acknowledgment segment. In the actual SACK proposal [10], there are also precise rules for determining which blocks should be included in the acknowledgment segment and in what order. A nondeterministic selection of blocks is a generalization that includes the selection of the blocks with the precise rules as a subset of its possibilities. Therefore, if our model of the protocol is safe, then using more precise rules is also safe.

The receiver passes data to the external user via the output action $\texttt{deliver}(\texttt{m})$, where m is the data part of a contiguous prefix of the receive buffer.

The internal action drop(b) results in a nondeterministically chosen sequence of data being dropped from rcv_buf. We include this action because in the proposal for the SACK mechanism [10], a receiver is allowed to drop SACKed data if it receives contiguous data for which it does not have enough space. The nondeterministic drop(b) action is a generalization of this possibility.

D. Complete Specification of SACK

An automaton for the selective acknowledgment mechanism is formed by the parallel composition of the automata for the sender, the receiver, and the channels. The channel from the receiver to the sender must take packets that are either only integers or packets that are a tuple of an integer and three SACK blocks. We define AckPkt to be this type. That is, **type** AckPkt $=$ Int $\cup$ **tuple of** Ack:Int, B1:Blk, B2:Blk, B3:Blk. Therefore, the composed automaton for SACK is

$$\texttt{Sack} \triangleq \texttt{S}||\texttt{R}||\texttt{C}_{\texttt{sr}}(\texttt{T}:\texttt{Seq[ByteInt]})||\texttt{C}_{\texttt{rs}}(\texttt{T}:\texttt{AckPkt}).$$

V. Proof of Safety

For the proof of safety, we use the formalization of simulations developed by Lynch and Vaandrager [8]. Let A be an automaton representing an implementation of a protocol and B be an automaton representing an abstract requirements of the protocol. If A and B have the same input and output actions, then a simulation from A to B is a relation between states of A and states of B such that certain conditions hold. The conditions that hold depend on whether we do a *forward* simulation (a special case of which is a *refinement mapping*) or a *backward* simulation. The simulation techniques have two general conditions. First, the start states of the two automata must be related in a certain way, and second, each step of the implementation must "simulate" some sequence of steps in the requirements. That is, for each step in the implementation, there must exist a sequence of steps in the requirements between states related by the simulation relation to the pre- and post-state of the implementation step, such that the sequence of requirements steps contains exactly the same external actions as the implementation step. This implies that traces of A are also traces of B. In this paper, we use a refinement mapping. We write $A \leq_R B$ if there exists a refinement mapping from A to B, and $A \leq_R B$ via r if r is such a mapping.

During the process of performing a simulation proof, sometimes a situation that is impossible to solve comes up, but then it turns out that the situation happens in an unreachable state. Since we only need to consider the steps of the implementation automaton which start in a reachable state, an *invariant* that rules out these "bad" states is found. Invariants are properties that are true of all reachable states.

A. Invariants for Sack

We formally verify that the SACK protocol satisfies the safety properties of reliable message delivery by defining a mapping from states of Sack to states of ReliableQ and then proving that it is a refinement mapping. However, before we proceed with the proof, we need to define some invariants for Sack. These invariants limit the states we need to consider during the simulation. The invariants that we present here capture some of the key insights as to why the protocol is reliable. The proofs for the invariants are straightforward by induction and can be found in Appendix II.

The first invariant shows the relationships between $\texttt{snd\_una}, \texttt{snd\_nxt}$, and the sequence numbers of the first and last elements of the retransmission buffer, $\texttt{retran\_buf}$.

Invariant 1:

1) If $\texttt{retran\_buf} \neq \{\}$
then $\texttt{snd\_una} = \texttt{head}(\texttt{get\_seq}(\texttt{retran\_buf}))$.
2) If $\texttt{retran\_buf} \neq \{\}$
then $\texttt{snd\_nxt} = \texttt{last}(\texttt{get\_seq}(\texttt{retran\_buf})) + 1$.
3) If $\texttt{retran\_buf} = \{\}$ then $\texttt{snd\_una} = \texttt{snd\_nxt}$.

The next invariant states that the elements of the retransmission buffer, of segments from the sender, and of the receive buffer, are sorted in ascending sequence number order. Additionally, the sequence numbers of elements of the retransmission buffer and of segments are contiguous.

Invariant 2:

1) If $\texttt{retran\_buf} \neq \{\}$
then for every element $(d_i, s_i, b_i) \in \texttt{retran\_buf}$,
except $\texttt{last}(\texttt{retran\_buf}), s_{i+1} = s_i + 1$.
2) If $\texttt{segment} \neq \{\}$
then for every element $(d_i, s_i) \in \texttt{segment}$,
except $\texttt{last}(\texttt{segment}), s_{i+1} = s_i + 1$.
3) If $\texttt{in\_transit}_{\texttt{sr}} \neq \{\}$
then for every $t \in \texttt{in\_transit}_{\texttt{sr}}$ and any $(d_i, s_i) \in p$
except $\texttt{last}(t)$ $s_{i+1} = s_i + 1$.
4) If $(d_i, s_i) \in \texttt{rcv\_buf}$ and $(d_j, s_j) \in \texttt{rcv\_buf}$ and $i < j$
then $s_i < s_j$.

Invariant 3 states that $\texttt{snd\_nxt}$ is always greater than the sequence number of any data in the retransmission buffer, the receiver buffer, or on $\texttt{C}_{\texttt{sr}}(\texttt{T})$, and that it is greater than or equal to $\texttt{rcv\_nxt}$. It also states that when the receive buffer is not empty, $\texttt{rcv\_nxt}$ is 1 plus the sequence number of the last contiguous piece of data in the receive buffer.

Invariant 3:

1) $\texttt{snd\_nxt} \geq \texttt{rcv\_nxt}$
and $\texttt{snd\_nxt} > \texttt{last}(\texttt{get\_seq}(\texttt{segment}))$
and $\texttt{snd\_nxt} > \texttt{last}(\texttt{get\_seq}(\texttt{rcv\_buf}))$
and $(\forall t \in \texttt{in\_transit}_{\texttt{sr}} : \texttt{snd\_nxt} > \texttt{last}(\texttt{get\_seq}(t)))$
and $(\forall q \in \texttt{in\_transit}_{\texttt{rs}} : \texttt{snd\_nxt} \geq \texttt{q})$
2) If $\texttt{rcv\_buf} \neq \{\}$
then $\texttt{rcv\_nxt} = \texttt{seq\_max\_con}(\texttt{rcv\_buf}) + 1$.
3) For any $t \in \texttt{in\_transit}_{\texttt{rs}}, \texttt{rcv\_nxt}$ is greater than or equal to the acknowledgment number (ack) of packet t.

The final invariant says that elements in buffers or in segments that have the same sequence number part also have the same data part.

Invariant 4: If $(d_h, s_h, b_h) \in \texttt{retran\_buf}$ and $(d_i, s_i) in \texttt{segment}$ and $(d_j, s_j) \in t$ for any $t \in \texttt{in\_transit}_{\texttt{sr}}$ and $(d_k, s_k) \in \texttt{rcv\_buf}$ then $s_h = s_i \Rightarrow d_h = d_i$ and $s_h = s_j \Rightarrow d_h = d_j$ and $s_h = s_k \Rightarrow d_h = d_k$ and $s_i = s_j \Rightarrow d_i = d_j$ and $s_i = s_k \Rightarrow d_i = d_k$ and $s_j = s_k \Rightarrow d_j = d_k$.

The conjunction of Invariants 1 through 4 is itself an invariant which we call I_S.

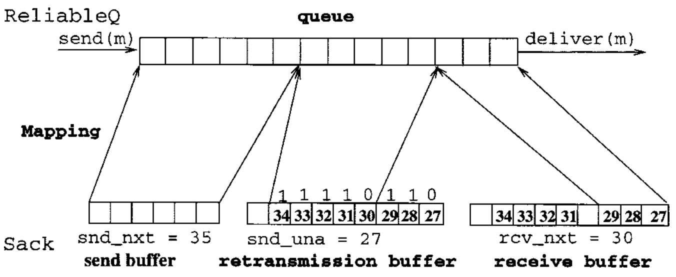


Fig. 8. Example of how the buffers in Sack relate to queue in ReliableQ by the refinement mapping.

B. Refinement Mapping

We define a function from *states*(Sack) to *states*(ReliableQ). Fig. 8 illustrates the mapping. It is formally defined below.

Definition 1 (Refinement Mapping R_{SR}): If $s \in \text{states}(\texttt{Sack})$, then define $R_{\text{SR}}(s)$ to be the state $u \in \text{states}(\texttt{ReliableQ})$ such that $u \cdot \texttt{queue} =$ concatenation of:

- $\texttt{l\_cprefix}(\texttt{get\_data}(\texttt{s} \cdot \texttt{rcv\_buf})))$
- $\texttt{delete}(\texttt{get\_data}(\texttt{s} \cdot \texttt{retran\_buf}), \texttt{s} \cdot \texttt{rcv\_nxt})$
- $s \cdot \texttt{send\_buf}$

The key observation that leads to a proof of safety for SACK is that the changes in the acknowledgment mechanism due to SACK are conservative in terms of safety. That is, no data is discarded, duplicated, or reordered because of the SACK mechanism that could not have been discarded, duplicated, or reordered with the cumulative acknowledgment mechanism. The fact that the mapping we define above ignores the additional information that the SACK mechanism provides reflects this insight.

For a refinement mapping from *states*(Sack) to *states*(ReliableQ), it is easy to see that $\texttt{send\_buf}$ in Sack should map to a suffix of the abstract queue. It is also clear that parts of the retransmission buffer and the receive buffer of Sack should map to the rest of the abstract queue. However, since some data may be in both buffers at the same time, the mapping has to be defined so that there is no duplicate data in the abstract queue. Also, since there is data in the receive buffer that may get dropped, this data cannot be included in the mapping. Therefore, in our mapping we use $\texttt{rcv\_nxt}$ as the demarcation point for data that is included in the mapping from both buffers. For the receive buffer, data with sequence numbers less than $\texttt{rcv\_nxt}$ are including in the mapping as a prefix of the abstract queue. This part of the $\texttt{rcv\_buf}$ is defined by the $\texttt{l\_cprefix}(\texttt{get\_data}(s \cdot \texttt{rcv\_buf})))$ operation. In the retransmission buffer, data with sequence numbers greater than or equal to $\texttt{rcv\_nxt}$ are included in the mapping. The $\texttt{delete}(\texttt{get\_data}(\texttt{s} \cdot \texttt{retran\_buf}), \texttt{s} \cdot \texttt{rcv\_nxt})$ denotes this section of the retransmission buffer.

C. Simulation of Steps

In this section, we prove that the mapping R_{SR} defined in the previous section is indeed a refinement mapping from the states of Sack to the states of ReliableQ. That is, we prove the following lemma.

Lemma 1: $\texttt{Sack} \leq_{\mathrm{R}} \texttt{ReliableQ}$ via R_{SR}

Proof: We prove that R_{SR} is a refinement mapping from Sack to ReliableQ with respect to I_S by showing that the two start states are related, and then showing that the steps of Sack simulate some sequence of steps of ReliableQ. That is, any behavior of Sack corresponds to some possible behavior of ReliableQ.

Base Case: In the start state s_0 of Sack, we have $s_0 \cdot \texttt{rcv\_buf}, s_0 \cdot \texttt{retran\_buf}$, and $s_0 \cdot \texttt{send\_buf}$ all being the empty sequence, $\{\}$. In the start state u_0 of ReliableQ, $u_0 \cdot \texttt{queue}$ is also the empty sequence.

Inductive Case: Assume $(s, a, s') \in \mathrm{Steps}(\texttt{Sack})$. Below, we consider cases based on a, and for each case we define a finite execution fragment α of ReliableQ such that $\mathrm{fstate}(\alpha) = R_{\mathrm{SR}}(s), \mathrm{lstate}(\alpha) = R_{\mathrm{SR}}(s')$, and $\mathrm{trace}(\alpha) = \mathrm{trace}(s, a, s')$. We use u and u' to denote $R_{\mathrm{SR}}(s)$ and $R_{\mathrm{SR}}(s')$, respectively.

$a = \texttt{send(m)}$*:* For this case, $\alpha = (u, a, u')$. In Sack, step (s, a, s') adds m to the back of send_buf. In ReliableQ, α adds m to the back of $s \cdot \texttt{queue}$, so the mapping is preserved after this step.

$a = \texttt{prepare-new-seg(s)}$*:* For this case, α is the empty step. Since prepare-new-seg(s) is an internal action, the traces of both steps are the same. Step (s, a, s') may take data from $s \cdot \texttt{send\_buf}$ and add it to the end of $s \cdot \texttt{retran\_buf}$. However, since this step does not change rcv_nxt and Invariant 3 tells us that $\texttt{snd\_nxt} \geq \texttt{rcv\_nxt}$, the mapping to $u' \cdot \texttt{queue}$ holds after this step.

$a = \texttt{send-pkt}_{\texttt{sr}}\texttt{(seg)}$*:* For this case, α is the empty step. Since $\texttt{send-pkt}_{\texttt{sr}}\texttt{(seg)}$ is an internal action of Sack, the trace of both steps are the same. Step (s, a, s) does not affect any of the variables of Sack involved with the mapping, so clearly the mapping is preserved after this step.

$a = \texttt{rcv-pkt}_{\texttt{rs}}\texttt{(ack)}$ *and* $a = \texttt{rcv-pkt}_{\texttt{rs}}\texttt{(ack, b1, b2, b3)}$*:* Again, α is the empty step. These steps may affect the mapping because they may cause the sender to discard the prefix of retran_buf with data that has sequence number less than ack. However, by Invariant 3 we know that $\texttt{rcv\_nxt} \geq \texttt{ack}$, so the part of the buffer that gets deleted is not included in the mapping. Thus, after step (s, a, s'), the mapping still holds.

$a = \texttt{prepare-retran-seg(s)}$ *and* $a = \texttt{reset-sack}$*:* For these cases, we again have α being the empty step. The steps formed with either of these actions clearly do not affect any of the variables of Sack involved with the mapping.

$a = \texttt{rcv-pkt}_{\texttt{sr}}\texttt{(seg)}$*:* For this step, the corresponding α is the empty step. The traces for both (s, a, s') and α are both clearly empty. Step (s, a, s') may affect the mapping because it may cause rcv_nxt to be updated and seg to be inserted in to $s \cdot \texttt{rcv\_buf}$. Therefore, we need to show that any additional data from $s' \cdot \texttt{rcv\_buf}$ that is now included in the mapping is precisely the data that is no longer included from $s' \cdot \texttt{retran\_buf}$ in the mapping. We know by Invariant 3 that $\texttt{snd\_nxt} \geq \texttt{rcv\_nxt}$ and by Invariant 1 we know that if $\texttt{retran\_buf} \neq \{\}$, then $\texttt{snd\_nxt} = \texttt{last(get\_seq(retran\_buf))} + 1$. Since $s \cdot \texttt{retran\_buf}$ and $s \cdot \texttt{snd\_nxt}$ are unaffected by this action, we know that if $s' \cdot \texttt{rcv\_nxt} > s \cdot \texttt{rcv\_nxt}$, then $s \cdot \texttt{retran\_buf} \neq \{\}$. Therefore, in the case where step (s, a, s') causes rcv_nxt to change, we know that $s \cdot \texttt{retran\_buf} \neq \{\}$. We also know by Invariant 2 that the elements of retran_buf have contiguous and sorted sequence numbers. Thus, since $s \cdot \texttt{snd\_nxt} = \texttt{last(get\_seq(}s \cdot \texttt{retran\_buf))} + 1$ and $s \cdot \texttt{snd\_nxt} \geq s' \cdot \texttt{rcv\_nxt}$, we know that the elements of $s \cdot \texttt{retran\_buf}$ not included in the mapping in state s' have exactly the same sequence numbers as the new elements of $s' \cdot \texttt{rcv\_buf}$ now included in the mapping. Since we know by Invariant 4 that bytes with the same sequence numbers are the same, we know the mapping holds in state s'.

$a = \texttt{send-pkt}_{\texttt{rs}}\texttt{(ack)}$ *and* $a = \texttt{send-pkt}_{\texttt{rs}}\texttt{(ack, b1, b2, b3)}$*:* For these steps, the corresponding α is the empty step. It is easy to see that these steps do not affect the mapping.

$a = \texttt{deliver(m)}$*:* For this case, $\alpha = (u, a, u')$. In Sack, this step removes m from the front of rcv_buf, but only if the sequence numbers of the bytes contained in m are less than rcv_nxt. Therefore, this block of data corresponds to a block of data at the front of $s \cdot \texttt{queue}$. Thus, after the step in ReliableQ, the resulting state is correct, as defined by mapping R_{SR}.

$a = \texttt{drop(b)}$*:* For this case, α is the empty step. Since drop(b) is an internal action of Sack, the traces are the same. Step (s, a, s') removes data from $s \cdot \texttt{rcv\_buf}$, but since the dropped data cannot be part of the contiguous prefix of $s \cdot \texttt{rcv\_buf}$, the removal of this data does not affect the mapping. ■

The fact that we have proven that R_{SR} is a refinement mapping from the states of Sack to the states of ReliableQ, coupled with the soundness of refinement mappings for showing trace inclusion leads to our main result. Thus, we get the following theorem.

Theorem 1: Sack safely implements ReliableQ. That is, $\mathrm{traces}(\texttt{Sack}) \subseteq \mathrm{traces}(\texttt{ReliableQ})$.

Proof: The theorem follows from the soundness of refinement mappings for proving trace inclusion [8], and from Lemma 1. ■

VI. Performance Analysis

In this section, we provide a formal analysis of the improved performance of SACK TCP versus TCP Reno, when more than one packet is lost from a window of data. We will examine an execution where no retransmitted or acknowledgment packet is dropped. Our analysis is for the worst-case behavior of the cumulative ACK policy measured in terms of latency to deliver packets in sequence for this type of execution. This worst case occurs when the lost packets are the first packets sent in the window.

For the proof of safety, we do not need to use time in the model, so it was not included. We used as simple a model as possible for the safety analysis. However, for our analysis of performance properties, we need to augment our formal model of SACK to include time. In order to compare the performances of TCP Reno and SACK TCP we also need to model the cumulative acknowledgment mechanism of TCP Reno. To model a sender and a receiver that use time, we use the general timed automaton (GTA) model [6] which extends the I/O Automaton model to include real time. In our modeling, we make the following assumptions: 1) the lower bound on packet transit time on a channel in either direction is δ; 2) when the sending of a packet is enabled, whether it is new data, a retransmission, or an acknowledgment, there is a lower bound of ϵ on the processing time needed to generate that packet; and 3) the retransmission timeout is greater than the time it takes for the sender to receive three duplicate ACKs when there is packet loss.

Given these timing assumptions, we can calculate the worst-case latency of packets in a window of data for TCP Reno and SACK TCP. In order to differentiate the performance benefit possible because of the additional information provided by SACK versus benefits that are due to the fact that TCP Reno is conservative in its retransmission strategy, we use the somewhat more aggressive retransmission strategy suggested by Hoe [3]. With Hoe's strategy, when an acknowledgment is received after the retransmission of the first missing packet, if there is other missing data, the acknowledgment of the first retransmitted packet indicates the next missing packet, which can then be immediately retransmitted. TCP Reno may actually time out before the second packet is retransmitted. Since the retransmission timeout period is typically much greater that the round-trip time, the performance of TCP Reno may actually be much worse in this situation than what we show in our analysis.

Because of space considerations, we do not include the models here. However, the complete models are available in [12]. The models, Reno_TCP and Sack_TCP, consist of four components, as does the previous model. These are $\mathtt{Reno\_S}(\text{sender}), \mathtt{Reno\_R}(\text{receiver}), \mathtt{C_{sr}}(\mathtt{T})$, and $\mathtt{C_{rs}}(\mathtt{T})$, for Reno_TCP, and $\mathtt{Sack\_S}(\text{sender}), \mathtt{Sack\_R}(\text{receiver}), \mathtt{C_{sr}}(\mathtt{T})$, and $\mathtt{C_{rs}}(\mathtt{T})$ for Sack_TCP.

A. Proofs of Latency Bounds

For the proofs of the next two propositions, we make the following assumptions: 1) at time t_0 the sender has a window of size N and also N segments of data in its send buffer; 2) at time t_0 the sender receives an ACK indicating it can send the first element in the send buffer; 3) the first $k \geq 1$ packets that the sender sends after time t_0 get lost; and 4) at least three packets are passed to the receiver after the loss of the first k packets. If the above assumptions hold, then for TCP Reno we have the following proposition.

Proposition 1: In an execution of Reno_TCP where the first k packets are lost in a data window, the lower bound for the delivery of the ith packet in the window is $(k+5)\epsilon + 3\delta + \min(i-1, k-1) \times 2(\delta+\epsilon)$ after time t_0.

Before we formally prove the proposition, we provide some intuition. Note that when we refer to the first k packets, it is not a reference to the actual sequence numbers of the packet, but to the order in which the packets were sent after time t_0. Also note that because an acknowledgment is received at time t_0 indicating that the first packet currently in the send buffer can be sent, any subsequent ACK indicating that that first element can be sent is a duplicate ACK. The ACK received at time t_0 could be the ACK of a synchronization packet or previous data packets. The basic idea for the proof is that three duplicate ACKs must be received before the first dropped packet is retransmitted, and each subsequent dropped packet is only retransmitted after the acknowledgment of the previously retransmitted packet. The proof carefully adds up the minimum time required for these actions by taking into account the lower bounds on packet delivery time δ and packet processing time ϵ.

Proof: The proof is by induction on $i \leq k$. The packets after the kth packet cannot be delivered before the kth packet is received, because the receiver can only pass contiguous data to the user. Thus, the proof for $i \leq k$ is sufficient to prove the proposition.

For the base case, we assume $i = 1$. With the Reno_S component of Reno_TCP, three duplicate ACKs must be received before the sender retransmits a packet. Therefore, if the first k packets are dropped, packet $k+3$ causes the third duplicate ACK. In the model of the sender, when each packet is sent, there must be a wait of at least ϵ before the next packet can be sent. Therefore, packet $k+3$ cannot be sent before time $t_0+(k+3)\epsilon$. In our model of the channel from the sender to the receiver, a packet that is placed on the channel cannot be removed before at least δ time has passed. Therefore, the earliest that packet $k+3$ can arrive at the receiver is at time $t_0 + (k+3)\epsilon + \delta$. The receiver model Reno_R needs at least ϵ time to generate a response, so because of the minimum δ delay on the transit time of the ACK packet on $\mathtt{C_{rs}}$, the earliest the sender can receive this duplicate ACK is at time $t_0 + (k+4)\epsilon + 2\delta$. The sender can then retransmit packet one at time $t_0 + (k+5)\epsilon + 2\delta$. The lower bound for the arrival time of this packet at the receiver is $t_0 + (k+5)\epsilon + 3\delta$, at which time it may be passed to the user.

For the inductive case, we assume that the proposition holds for packet $j \leq k-1$ and now show that it holds for packet $j+1$. Since Reno_R can only pass contiguous data to the user, we know it cannot pass packet $j+1$ to the user until it receives packet j. By the inductive hypothesis, packet j is first received at time $(k+5)\epsilon + 3\delta + \min(j-1, k-1) \times 2(\delta+\epsilon)$ after time t_0. Since Reno_R can only indicate the first missing packet it needs, the value of the acknowledgment number is less than $j+1$ until after packet j is received. If the receiver generates this acknowledgment on receiving packet j, this acknowledgment arrives at the Reno_S at time $t_0 + (k+5)\epsilon + 3\delta + \min(j-1, k-1) \times 2(\delta+\epsilon) + \epsilon + \delta$. When the sender receives this acknowledgment, it takes at least time ϵ to retransmit packet $j+1$, which takes at least time δ to arrive at the receiver. Thus, packet $j+1$ arrives at the receiver at time $t_0 + (k+5)\epsilon + 3\delta + \min(j-1, k-1) \times 2(\delta+\epsilon) + \epsilon + \delta + \epsilon + \delta$, which is equal to $(k+5)\epsilon + 3\delta + \min(j, k-1) \times 2(\delta+\epsilon)$ after time t_0. ■

The next proposition analyzes the performance of Sack_TCP in the same situation as the analysis for Proposition 1, that is, with the same assumptions listed above. The proposition proves that the SACK mechanism allows better performance when $k > 1$ than is possible with cumulative ACKs. For $k = 1$, both mechanisms have the same performance.

Proposition 2: In an execution of Sack_TCP, where the first k packets are lost in a data window, the lower bound for the delivery of the ith packet in the window is $(k+5)\epsilon + 3\delta + \min(i-1, k-1) \times \epsilon$ after time t_0.

As in the proof of Proposition 1, the proof here is simply a careful accounting of the minimum time required for a set of actions. For SACK, after the three duplicate ACKs are received, not only the first but all dropped packets can be retransmitted.

Proof: The proof is by induction on $i \leq k$. As is the case for TCP Reno, this is sufficient because packets with sequence number $i > k$ cannot be delivered before the kth packet is delivered.

For the base case, we know that with the $\texttt{Sack\_S}$ component of $\texttt{Sack\_TCP}$, three duplicate ACKs must be received before the sender retransmits a packet. Thus, at least three packets must have arrived at $\texttt{Sack\_R}$ to generate three duplicate ACKs. The third duplicate ACK must contain a SACK block that indicates that packets $k+1$ through $k+3$ have been received. Therefore, when the sender receives the third duplicate ACK at time $t_0 + (k+4)\epsilon + 2\delta$, it knows that packet one through k are lost. Thus, it can send a retransmission of packet one at time $t_0 + (k+5)\epsilon + 2\delta$, which because of the lower bound on delivery time of $\texttt{C}_{\texttt{sr}}$ arrives at the receiver at time $(k+5)\epsilon + 3\delta$ after time t_0.

For the inductive case, we assume the proposition holds for $j \leq k-1$, and we now show that it holds for $j+1$. We know that the SACK block sent by $\texttt{Sack\_R}$ lets $\texttt{Sack\_S}$ know all the packets that are missing in this situation when the first dropped packet is retransmitted. Therefore, since $\texttt{Sack\_S}$ has a lower bound of ϵ between the sending of packets, packet $j+1$ may be retransmitted at period of ϵ after packet j was retransmitted. Since by the inductive hypothesis, packet j arrives at the receiver at time $t_0 + (k+5)\epsilon + 3\delta + \min(j-1, k-1) \times \epsilon$. Because of the delay of $\texttt{C}_{\texttt{sr}}(\texttt{T})$, it must have been retransmitted at time $t_0 + (k+5)\epsilon + 2\delta + \min(j-1, k-1) \times \epsilon$. Therefore, packet $j+1$ can be retransmitted at time $t_0 + (k+5)\epsilon + 2\delta + \min(j-1, k-1) \times \epsilon + \epsilon$, which means the earliest it can arrive at the receiver is at time $(k+5)\epsilon + 3\delta + \min(j, k-1) \times \epsilon$ after time t_0. ■

The propositions show that when more than one packet is dropped in a window of data, the SACK mechanism can improve latency by a factor proportional to $(k-1) \times (2\delta + \epsilon)$. The time period $2\delta + \epsilon$ is essentially a lower bound on round-trip time, so the improvement is proportional to $\mathrm{RTT} * (k-1)$. This latency results in idleness on the channel for the TCP connection and a consequent loss in throughput. A similar result is implied in [1] which shows that TCP implementations without SACK must either retransmit at most one dropped packet per round-trip time, or retransmit packets that might have already been successfully received.

B. Other Performance Gains

In this section, we show that SACK can improve performance relative to ACK even when window sizes are small and/or when acknowledgment packets get lost. The performance gain can be realized in these situations if the sender uses the additional information provided by the SACK blocks to further improve the retransmission strategy.

1) Improved Retransmission Strategy: The underlying retransmission strategy at the sender is to wait for a reliable indication that a segment has been lost before retransmitting the missing segment. One such indication is a retransmission timeout. The fast retransmit algorithm of TCP Reno uses three duplicate ACKs as a much quicker indication of a lost packet. The number three is chosen for engineering reasons. That is, one or two segments may be reordered, but three reordered segments is very unlikely.

With SACK blocks, the sender can still wait for three duplicate ACKs, but SACK blocks provide another reliable indicator of a missing segment. This indicator is a SACK block that records a segment with sequence number sufficiently larger than an unacknowledged segment. By the same engineering criterion that says three duplicate ACKs is an indication of loss and not reordering, we define "sufficiently larger" to be at least $(3 \times \mathrm{MSS})$. Thus, the sender may retransmit the first unacknowledged segment after it receives three duplicate ACKs, or it may retransmit any unacknowledged segment that is at least $(3 \times \mathrm{MSS})$ less than the largest sequence number of a segment acknowledged in a SACK block. This strategy is not as aggressive as possible, but is consistent with the current TCP strategy which allows for the possibility of some reordering of packets. Because one SACK packet can provide enough information for the sender to declare a segment lost, this strategy allows the sender to recover from losses even if the receiver did not receive the segments necessary to generate three duplicate ACKs.

In the previous paragraph, we propose a retransmission strategy at the sender based on waiting for an indication that a segment with large enough sequence number has been received. This is basically the same as the strategy used by by Mathis and Mahdavi [9]. In their algorithm, they use a new variable, $\texttt{snd\_fack}$, at the sender. This new variable is updated to reflect the forwardmost data held by the receiver, and is derived from the SACK blocks. Their strategy is to have the sender wait until $\texttt{snd\_fack-snd\_ack} > (3 \times \mathrm{MSS})$ or three duplicate ACKs are received. We contend that the correct strategy for when to retransmit is to wait until $\texttt{snd\_fack-snd\_ack} \geq (3 \times \mathrm{MSS})$. That is, the inequality does not have to be strict. Furthermore, the additional test for three duplicate ACKs is not needed because whenever there are three duplicate ACKs, $\texttt{snd\_fack-snd\_ack} \geq (3 \times \mathrm{MSS})$ is also true. However, $\texttt{snd\_fack-snd\_ack} \geq (3 \times \mathrm{MSS})$ may be true before three duplicate ACKs are received. Thus, the $\texttt{snd\_fack-snd\_ack} \geq (3 \times \mathrm{MSS})$ test subsumes the three duplicate ACKs test.

2) Small Window Sizes and Lost Acknowledgments: The fact that we use $\texttt{snd\_fack-snd\_ack} \geq (3 \times \mathrm{MSS})$ as the criterion for determining when to retransmit means that SACK may help speed recovery even for cases where the window size is so small that three duplicate ACKs may not be generated when there are lost segments. The smallest window size where SACK may be beneficial is four. Fig. 9 illustrates such a situation. We assume the sender has already received an ACK asking for packet 26 prior to the beginning of the figure. In the figure, the first two packets are lost from a window of size four. Therefore, only two duplicate ACKs are generated. However, the SACK block indicates that packet 29 has been received, so the first lost packet is recovered before a timeout. The second packet is recovered when the acknowledgment for the first packet is received. In the same situation with TCP Reno (Fig. 10), the sender has to timeout before retransmitting the first lost packet.

The loss of an acknowledgment packet in the cumulative ACK mechanism for transmission with small window sizes can also cause performance degradation. Again, the problem occurs because the sender needs to wait for enough duplicate ACKs before it can decide that a packet is lost and retransmit

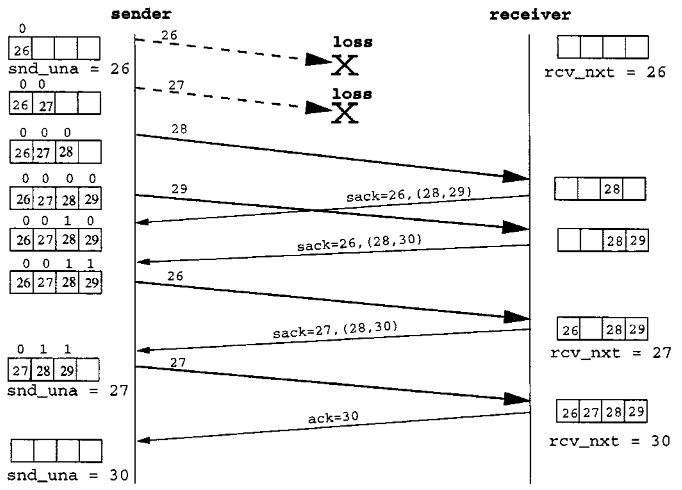


Fig. 9. How the SACK protocol enables the sender to quickly recover from losses even with a window size of four.

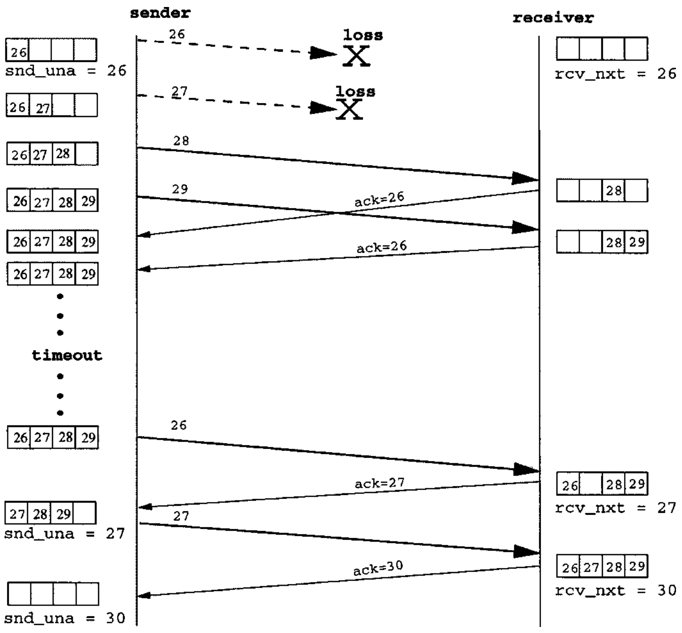


Fig. 10. With the cumulative ACK mechanism, the sender has to wait for a timeout to recover from the losses.

packets. In the case of SACK, even though some number of the (S)ack packets are lost, when the first SACK arrives at the sender, there is enough information to say at least three packets have been received after the first hole in the sequence number space. For example, in the situation described above where the window size is four, if only the first data packet gets lost, but the acknowledgment of the second data packet also gets lost, then for cumulative ACK we again have the situation where the sender has to timeout before it can retransmit the dropped packet. In contrast, with SACK, the sender can recover on receipt of a single SACK, rather than having to receive three ACKs.

VII. Conclusion

In this paper, we presented a formal specification of the TCP selective acknowledgment option. Our model succinctly and completely captures the important properties of this mechanism. Using our model, we are able to formally verify that the SACK mechanism for TCP is safe in that it does not violate the properties of reliable end-to-end message delivery. We extended the basic specification to include time using the GTA model [6]. We also used the GTA formalism to model the cumulative ACK mechanism with fast retransmit. Using these models, we proved that in certain worst-case scenarios, where there are multiple packets lost consecutively, SACK can reduce latency by a factor proportional to the round-trip time times the number of packets lost.

Using our formal model, we were also able to explore ways to use the additional information provided by SACK to improve the retransmission strategy of TCP. We presented a retransmission strategy that allows the sender to retransmit whenever the fast retransmit strategy allows a retransmission. However, our strategy also allows the sender to retransmit in situations where a fast retransmit may not occur. In these situations, our retransmission strategy can prevent timeouts at the sender, and therefore, can improve TCP performance significantly. In particular, our strategy is useful if window sizes are small and/or if acknowledgment packets are lost.

Appendix I
Notation and Basic Definitions

We model buffers and data segments as lists. We also use the terms "sequence" or "queue" synonymously with list. We assume lists are finite and we write a list l consisting of the elements $e_1, e_2, \ldots, e_n$ as follows: $l = \langle e_1, e_2, \ldots, e_n \rangle$. We denote the empty list by $\{\}$.

A. General List Operations

The length of a list $l = \langle e_1, e_2 \ldots, e_n \rangle$, written $|l|$, is defined as $|l| \triangleq n$. If $l = \langle e_1, e_2 \ldots, e_n \rangle$ is nonempty, define

$$\mathtt{head}(l) \triangleq e_1$$
$$\mathtt{last}(l) \triangleq e_n$$
$$\mathtt{tail}(l) \triangleq \langle e_2, e_3, \ldots, e_n \rangle.$$

For $0 \leq k < n$, define

$$\mathtt{tail}(l, k) \triangleq \langle e_{k+1}, e_{k+2}, \ldots, e_n \rangle.$$

Single elements can appended to lists. For $l = \langle e_1, \ldots, e_n \rangle$ and element e_{n+1}, define

$$l \vdash e_{n+1} \triangleq \langle e_1, \ldots, e_n, e_{n+1} \rangle.$$

Concatenation of two lists l_1 and l_2 is written $l_1 || l_2$ or sometimes $l_1 l_2$. If $l_1 = \langle e_1, \ldots, e_n \rangle$ and $l_2 = \langle e_{n+1}, e_{n+2} \ldots, e_m \rangle$, then define

$$l_1 || l_2 \triangleq \langle e_1, \ldots, e_n, e_{n+1}, e_{n+2} \ldots, e_m \rangle.$$

If $l = \langle e_1, e_2, \ldots, e_n \rangle$ is nonempty, define the set of subsequences as

$$\mathtt{subseqs}(l) \triangleq \{\langle e_i, e_{i+1}, \ldots, e_{i+j} \rangle \mid 1 \leq i \leq n \text{ and } 0 \leq j \leq n - i\}.$$

If $l = \langle e_1, e_2, \ldots, e_n \rangle$ is nonempty, define

$$\texttt{prefixes}(l) \triangleq \{\{\}, \langle e_1 \rangle, \langle e_1, e_2 \rangle, \ldots \langle e_1, e_2, \ldots, e_n \rangle\}.$$

The operator `prefix` takes a list l and a number k, and returns a prefix of length k, or if the list has fewer than k elements, it returns all the elements of the list. For $l = \langle e_1, e_2, e_3, \ldots, e_n \rangle$, define

$$\texttt{prefix}(l, k) \triangleq \begin{cases} \langle e_1, e_2, \ldots, e_k \rangle & \text{if } k \le n \\ \langle e_1, e_2, \ldots, e_n \rangle & \text{if } k > n. \end{cases}$$

B. Operations on Lists With Special Elements

Let each element of the list be of the form (d_i, s_i) or (d_i, s_i, b_i), where d_i is of some data type, s_i is a sequence number, and b_i is a Boolean flag. The `delete` operator takes a list l, where the integer parts of the elements are assumed to be sorted in ascending order and an integer k and removes all the elements with integer parts strictly less than k. That is, given $l = \langle (d_1, s_1), (d_2, s_2), \ldots, (d_n, s_n) \rangle$, define

$$\texttt{delete}(l, k) \triangleq \begin{cases} l & \text{if } k \le s_1, \\ \{\} & \text{if } k > s_n, \\ \langle (d_i, s_i), (d_{i+1}, s_{i+1}), \ldots, (d_n, s_n) \rangle & \\ \quad \text{where } i = \min(1 \ldots n) \text{ such that } s_i \ge k & \text{otherwise.} \end{cases}$$

For lists with elements of type (d_i, s_i, b_i), we have the obvious equivalent definition. Another version of the operator takes the list and two integers, k_1 and k_2, but is defined only if $s_i \le k_1 \le k_2 \le s_n$, that is

$$\texttt{delete}(l, k_1, k_2) \triangleq \langle (d_1, s_1), (d_2, s_2), \ldots, (d_{k_2}, s_{k_2}), \ldots, (d_n, s_n) \rangle.$$

The `inst` operator inserts a list of sorted contiguous elements into another sorted list. Let $l = \langle e_1, e_2, \ldots \rangle$ where $e_i = (d_i, s_i)$ and $l' = \langle e'_1, e'_2, \ldots, e'_m \rangle$ where $e'_i = (d'_i, s'_i)$. Let *overlap* be the set of the elements of l' where the integer part of the element is equal to the integer part of an element in l, that is, $\text{overlap} = \{(d'_i, s'_i) \mid (d_j, s_j) \in l \wedge s_j = s'_i\}$, and let $t = |l| + m - |\text{overlap}|$. Then define

$$\texttt{inst}(l, l') \triangleq \langle e''_1, e''_2, \ldots e''_t \rangle$$

where $(\forall i \forall j : i < j \Rightarrow s''_i < s''_j) \wedge (\exists u \text{ s.t. } (e''_u = e'_1) \wedge (\forall v : 0 < v \le m \Rightarrow e''_{u+v} = e'_{1+v}) \wedge (\forall h : h < u \Rightarrow e''_h = e_h) \wedge (\forall k : (u + m < k \le t) \Rightarrow (\exists l \text{ s.t. } e''_k = e_l)))$.

Given $l = \langle (d_1, s_1), (d_2, s_2), \ldots, (d_n, s_n) \rangle$, or $l = \langle (d_1, s_1, b_1), (d_2, s_2, b_2), \ldots, (d_n, s_n, b_n) \rangle$, define

$$\texttt{get\_data}(l) \triangleq \langle d_1, d_2, \ldots, d_n \rangle$$
$$\texttt{get\_seq}(l) \triangleq \langle s_1, s_2, \ldots, s_n \rangle.$$

Given $l = \langle (d_1, s_1, b_1), (d_2, s_2, b_2), \ldots, (d_n, s_n, b_n) \rangle$, define

$$\texttt{remove\_flag}(l) \triangleq \langle (d_1, s_1), (d_2, s_2), \ldots, (d_n, s_n) \rangle.$$

Given $l = \langle (d_1, s_1, b_1), (d_2, s_2, b_2), \ldots, (d_n, s_n, b_n) \rangle$, define

$$\texttt{unsack}(l) \triangleq \langle (d_1, s_1, 0), (d_2, s_2, 0), \ldots, (d_n, s_n, 0) \rangle.$$

Given $l = \langle d_1, d_2, \ldots, d_n \rangle$, and sequence number k, define

$$\texttt{enum}(l, k) \triangleq \langle (d_1, k), (d_2, k+1), \ldots, (d_n, k+n-1) \rangle.$$

If $l = \langle (d_1, s_1), (d_2, s_2), \ldots, (d_n, s_n) \rangle$, define

$$\texttt{init\_flag}(l) \triangleq \langle (d_1, s_1, 0), (d_2, s_2, 0), \ldots, (d_n, s_n, 0) \rangle.$$

Let $l = \langle (d_1, s_1), (d_2, s_2), \ldots, (d_n, s_n) \rangle$, where $\forall i \forall j : i < j \Rightarrow s_i < s_j$. For the definition, please see the equation at the bottom of the page. The largest contiguous prefix of a list l, written $\texttt{l\_cprefix}(l)$, is the contiguous prefix with the most elements. Formally

$$\texttt{l\_cprefix}(l) \triangleq l' \in \texttt{cprefixes}(l) \text{ s.t. } \forall l'' : l'' \in \texttt{cprefixes}(l) \wedge l'' \ne l', |l'| > |l''|.$$

The `seq_max_con` operator returns the sequence number of the last element of the largest contiguous prefix of a list l. That is, for a list l, define

$$\texttt{seq\_max\_con}(l) = \texttt{last}(\texttt{get\_seq}(\texttt{l\_cprefixes}(l))).$$

The `set_sack` operator takes a list and three blocks of type $(\texttt{Int} \times \texttt{Int})$ where the first element of the pair is less than or equal to the second element. Given $l = \langle (d_1, s_1, b_1), (d_2, s_2, b_2), \ldots, (d_n, s_n, b_n) \rangle$, define

$$\texttt{set\_sack}(l, (k_1, k'_1), (k_2, k'_2), (k_3, k'_3)) \triangleq \langle (d_1, s_1, b'_1), (d_2, s_2, b'_2), \ldots, (d_n, s_n, b'_n) \rangle$$

where $\forall j : ((k_1 \le j < k'_1) \vee (k_2 \le j < k'_2) \vee (k_3 \le j < k'_3)) \Rightarrow b'_j = 1$ and $\forall h : ((h < k_1 \vee h \ge k'_1) \wedge (h < k_2 \vee h \ge k'_2) \wedge (h < k_3 \vee h \ge k'_3)) \Rightarrow b'_h = b_h$.

The `holes` operator takes a list with elements of the form (d_i, s_i, b_i) and returns the set of subsequences of the list where every element has $b_i = 0$. Formally

$$\texttt{holes}(l) \triangleq \{\langle e_i, e_{i+1}, \ldots, \rangle \in \texttt{subseqs}(l) \mid \forall e_i : b_i = 0\}.$$

The `nxt_unsacked` operator takes a list with elements of the form (d_i, s_i, b_i) and a sequence number m. Given $l = \langle (d_1, s_1, b_1), (d_2, s_2, b_2), \ldots, (d_n, s_n, b_n) \rangle$, define

$$\texttt{nxt\_unsacked}(l, m) \triangleq \begin{cases} (d_i, s_i) & \text{if } \exists i \text{ s.t. } b_i = 0 \wedge s_i > m \wedge \\ & \qquad \forall j \text{ s.t. } s_j > m \wedge s_j < s_i : b_j = 1 \\ \text{null} & \text{otherwise.} \end{cases}$$

The `blocks` operator takes a list with elements of type (d_i, s_i) and returns the set of all contiguous and isolated blocks of data.

$$\texttt{cprefixes}(l) \triangleq \begin{cases} \texttt{prefixes}(l) & \text{if } \forall e_i, s_i + 1 = s_{i+1} \\ \texttt{prefixes}(\texttt{prefix}(l, k)) & \text{if } \exists k \text{ s.t. } 1 \le i < k, s_i + 1 = s_{i+1} \wedge s_{k+1} - s_k > 1. \end{cases}$$

Each block is represented by a pair of type $(\mathtt{Int} \times \mathtt{Int})$. That is, given $l = \langle (d_1, s_1), (d_2, s_2), \ldots, (d_n, s_n) \rangle$, where $\forall i \forall j : i < j \Rightarrow s_i < s_j$, define

$$\begin{aligned}\mathtt{blocks}(l) \triangleq \{(k_i, k'_i) \,|\, \forall i : k_i \neq s_1 \wedge k_i < k'_i \wedge (\nexists h : s_h \\ = k'_i \vee s_h = k_i - 1) \wedge \exists j \text{ s.t. } (k_i = s_j) \\ \wedge ((\forall f : j \leq f < k'_i) \Rightarrow s_{f+1} = s_f + 1)\}.\end{aligned}$$

Appendix II Proof of Invariants

We use the standard inductive technique for proving the invariants. That is, we show that the invariants hold for the start states and then show that for every step (s, a, s') if the invariant holds in state s, then it also holds in state s'. The invariants are typically of the form "if P (premise) then C (consequence)", where P might be true. Therefore, we only need to consider actions that could cause P to go from false to true or could cause C to go from true to false. We call these actions critical actions.

Proof of Invariant 1: In the start state $\mathtt{retran\_buf} = \{\}$ and $\mathtt{snd\_una} = \mathtt{snd\_nxt}$, so the invariant holds for the base case. The inductive steps for Part 1 and Part 2 of this invariant are straightforward. We now consider the critical actions for Part 3.

$\mathtt{prepare\text{-}new\text{-}seg(s)}$*:* This action may cause the consequence to go from true to false. However, $\mathtt{retran\_buf} \neq \{\}$.

$\mathtt{rcv\text{-}pkt}_{\mathtt{rs}}\mathtt{(ack)}$ *and* $\mathtt{rcv\text{-}pkt}_{\mathtt{rs}}\mathtt{(ack,\ b1,\ b2,\ b3)}$*:* These actions may cause the premise of Part 3 to go from false to true if $\mathtt{ack} \leq \mathtt{snd\_nxt}$ and $\mathtt{ack} > \mathtt{last}(\mathtt{get\_seq}(s \cdot \mathtt{retran\_buf}))$. We know that $s \cdot \mathtt{snd\_nxt} = \mathtt{last}(\mathtt{get\_seq}(s \cdot \mathtt{retran\_buf})) + 1$ from Part 2 of this invariant. Thus, since after this step $s' \cdot \mathtt{snd\_una} = \mathtt{ack}$, if $s' \cdot \mathtt{retran\_buf} = \{\}$ then $s' \cdot \mathtt{snd\_una} = s' \cdot \mathtt{snd\_nxt}$. ∎

Proof of Invariant 2: In the start state $\mathtt{retran\_buf}, \mathtt{rcv\_buf}$, and $\mathtt{in\_transit}_{sr}$ are all empty, so the invariant holds in this state.

$\mathtt{prepare\text{-}new\text{-}seg(s)}$*:* This action is critical as it may cause the premise to go from false to true. By the definition of the enum operator it is clear that Part 1 holds after this step. For the case where $s \cdot \mathtt{retran\_buf} \neq \{\}$, this action does not cause the consequence to become false, because Part 2 of Invariant 1 tells us that in this situation $s \cdot \mathtt{rcv\_nxt} = \mathtt{last}(\mathtt{get\_seq}(s \cdot \mathtt{retran\_buf})) + 1$. Part 2 of the invariant clearly follows from Part 1, and it is easy to see that Part 3 follows from Part 2. Part 4 follows from Part 3 and the definition of the $\mathtt{inst}$ operator. ∎

Proof of Invariant 3: In the start state $\mathtt{retran\_buf} = \mathtt{segment} = \mathtt{rcv\_buf} = \{\}, \mathtt{rcv\_nxt} = \mathtt{snd\_nxt}$, and $\mathtt{in\_transit}_{\mathtt{sr}} = \mathtt{in\_transit}_{\mathtt{rs}} = \emptyset$, so the base case is true.

For the inductive step of Part 1, since the premise is true, we need to show that there are no critical actions. Using Invariants 2 and 1, it is straightforward to see that no action causes the consequences of Part 1 to become false. The inductive cases for Parts 2 and 3 are straightforward. ∎

Proof of Invariant 4: In the start state $\mathtt{retran\_buf} = \mathtt{segment} = \mathtt{rcv\_buf} = \{\}$, and $\mathtt{in\_transit}_{\mathtt{sr}} = \emptyset$, so the invariant holds for the base case.

$\mathtt{prepare\text{-}retran\text{-}seg(s)}$*:* This step may cause the premise of the invariant to go from false to true by adding elements to $s \cdot \mathtt{retran\_buf}$ and by assigning a new value to $s \cdot \mathtt{segment}$. However, by definition of the enum operator we know that $s' \cdot \mathtt{segment}$ has sequence numbers starting from $s \cdot \mathtt{rcv\_nxt}$ and from Invariants 3 and 2 we know that in state s, there are no elements of $s \cdot \mathtt{retran\_buf}, s \cdot \mathtt{rcv\_buf}$ or any segment $t \in s \cdot \mathtt{in\_transit}$ with sequence number greater than or equal to $s \cdot \mathtt{snd\_nxt}$.

$\mathtt{send\text{-}pkt}_{\mathtt{sr}}\mathtt{(seg)}$ *and* $\mathtt{prepare\text{-}retran\text{-}seg(s)}$*:* By the inductive hypothesis, the invariant holds in state s, so it clearly holds in state s' after these actions.

$\mathtt{rcv\text{-}pkt}_{\mathtt{sr}}\mathtt{(seg)}$*:* Since by the inductive hypothesis any element of $s \cdot \mathtt{seg}$ with the same sequence number as an element of $s \cdot \mathtt{rcv\_buf}, s \cdot \mathtt{retran\_buf}$, or $s \cdot \mathtt{segment}$ has the same data item, when this segment gets inserted into $s \cdot \mathtt{rcv\_buf}$, the invariant holds in state s'. ∎

Acknowledgment

The authors would like to thank S. Floyd for many valuable comments on an earlier draft of this paper, and M. Merrit and N. Klarlund for insightful discussions on the formal proof of safety. They would also like to thank the anonymous reviewers whose comments substantially improved the paper.

References

[1] K. Fall and S. Floyd, "Simulation-based comparisons of Tahoe, Reno, and SACK TCP," *Comput. Commun. Rev.*, vol. 26, no. 3, pp. 5–21, July 1996.

[2] S. J. Garland and N. A. Lynch, "The IOA language and toolset: Support for designing, analyzing, and building distributed systems," Mass. Inst. of Technol., Cambridge, Tech. Rep. MIT/LCS/TR-762, Aug. 1998.

[3] J. Hoe, "Improving the start-up behavior of a congestion control scheme for TCP," in *Proc. ACM SIGCOMM Symp. Communications Architectures and Protocols*, Aug. 1996, pp. 270–280.

[4] V. Jacobson, "Congestion control and avoidance," in *Proc. ACM SIGCOMM Symp. Communications Architectures and Protocols*, 1988, pp. 314–329.

[5] ——, Modified TCP congestion avoidance algorithm, 1990. E-mail to the end2end-interest mailing list.

[6] N. Lynch, *Distributed Algorithms*. San Mateo, CA: Morgan Kaufmann, 1996.

[7] N. Lynch and M. Tuttle, "An introduction to input/output automata," *CWI Quart.*, vol. 3, no. 2, Sept. 1989.

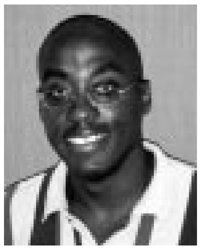
[8] N. Lynch and F. Vaandrager, "Forward and backward simulations—Part I: Untimed systems," *Inf. Comput.*, vol. 121, no. 2, pp. 214–233, Sept. 1995.

[9] M. Mathis and J. Mahdavi, "Forward acknowledgment: Refining TCP congestion control," in *Proc. ACM SIGCOMM Symp. Communications Architechures and Protocols*, Aug. 1996, pp. 281–291.

[10] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow, "TCP selective acknowledgment options,", Internet RFC-2018, Oct. 1996.

[11] M. Smith, "Formal verification of TCP and T/TCP," Ph.D. dissertation, Mass. Inst. of Technol., Cambridge, 1997.

[12] M. A. Smith and K. K. Ramakrishnan. Formal specification and verification of safety and performance of TCP selective acknowledgment. [Online]. Available: http://www.bell-labs.com/~massmith/papers/sack.html



Mark A. Smith was born in Jamaica. He received the B.Sc. degree from Brooklyn College of the City University of New York in 1989, and the S.M. and Ph.D. degrees from the Massachusetts Institute of Technology, Cambridge, in 1993 and 1997, respectively.

He is currently a Member of Technical Staff at Bell Labs, Lucent Technologies, Murray Hill. NJ. His research interests are distributed systems, network protocols, and Internet management problems.

Dr. Smith is a member of the Association for Computing Machinery.

K. K. Ramakrishnan (M'83) received the B.S. degree in electrical engineering from Bangalore University, India, in 1976, the M.S. degreee in automation from the Indian Institute of Science, India, in 1978, and the Ph.D. degree in computer science from the University of Maryland, Baltimore, in 1983.

He is a Founder and Vice President at TeraOptic Networks, Inc., Sunnyvale, CA. Until recently, he was a Technology Leader at AT&T Labs Research. From 1983 to 1994, he was with Digital Equipment Corporation working on network architecture and performance. His research interests are in design and performance of computer and communication network protocols and algorithms.

Dr. Ramakrishnan has been an editor for the IEEE/ACM TRANSACTIONS ON NETWORKING and the *IEEE Network Magazine*.