Formal Verification of Safety and Performance Properties of TCP Selective Acknowledgment

Mark A. Smith K. K. Ramakrishnan

AT&T Labs Research
180 Park Ave.
Florham Park, NJ 07932
E-mail: {mass,kkrama}@research.att.com

Abstract

We present a formal proof that the selective acknowledgment (SACK) mechanism that is being proposed as a new standard option for TCP [11] does not violate the safety properties 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. With selective acknowledgment, non-contiguous blocks of data can be acknowledged, and the sender only has to retransmit data that is actually lost. The proposed mechanism for implementing the SACK option for TCP is sufficiently complicated that it is not obvious that it is indeed safe. Because this mechanism is being proposed as a new standard for TCP, we think it is important to formally verify its safety properties.

We first present a formal automaton model of the SACK protocol. We then verify that SACK is indeed safe. The verification is done by first defining a simple specification of the required safety properties. the protocol is supposed to satisfy. We then use invariant assertion and simulation techniques to show the protocol indeed satisfies these properties.

Using the model we also show that SACK can improve the time it takes for the sender to recover from multiple packet losses, compared to the cumulative ACK protocol. Since there is additional information at the sender, SACK can save a round-trip time which the cumulative ACK mechanism has to wait before retransmitting subsequent packets lost after the very first loss.

1. Introduction

TCP is the most widely used transport protocol in the Internet today. It offers applications the semantics of a reliable, flow-controlled channel. The 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.

Selective Acknowledgments (SACK) [11] 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. SACK may offer a performance improvement, especially when multiple packets are lost in a round-trip time(RTT). The new functionality that SACK introduces is the potential for earlier recovery from loss of multiple packets. This also may result in higher throughput because the more severe congestion recovery mechanisms are not invoked. 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 $RTT * (k - 1)$, where k is the number of packets lost in a window.

The SACK mechanism includes sufficient additional complexity that we believe formally examining if it operates correctly or not is important. It is not obvious from reading the specification [11] that it satisfies the safety properties of the ACK mechanism. Simulation experiments have been done to understand the performance improvement [2]. While these lend confidence that the protocol operates as expected, simulations do not help ensure that the protocol is correct. Therefore, we feel that a formal verification of the safety properties of the mechanism is useful. We use invariant assertion and simulation (refinement) techniques to verify SACK. 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. We use these methods to show that the external behaviors of TCP with the SACK option, which we refer to as just 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 in [10] by Lynch and Vaandrager. The methodology is developed in the context of a very simple and general automaton. Simulation techniques are known to be quite useful in the verification of concurrent systems, and other researchers use this method in their work [7, 1, 12, 15].

The key property of correctness we focus on is that of safety of the SACK protocol: that data from the sender is not lost, duplicated or received out of order. As described in [11] data is acknowledged in a SACK block by the receiver, may be later dropped from the buffer of the receiver. However, the key observation is that this data remains in the retransmission buffer at the sender until it is

acknowledged with the regular cumulative acknowledgment mechanism. When a SACK is received for a segment, the sender only marks it with a SACK flag. The data segment remains in the sender's retransmission buffer until the cumulative acknowledgment is received. The SACK flags get reset when a retransmission timeout expires, so that the data is eventually retransmitted.

In the next section we present an informal description of both the cumulative ACK and SACK mechanisms. A brief description of the automaton model along with the abstract specification of the reliable message delivery problem is presented in Section 3. In Section 4 we present the formal model for the SACK mechanisms, and Section 5 contains the proof of the improved behavior possible with SACK relative to cumulative ACK. Section 6 has the proof of safety, and we conclude in Section 7.

2. Informal description of the acknowledgment mechanisms

Before we present the formal abstract specification for reliable message delivery and the formal model for SACK, we describe the cumulative ACK mechanism and the SACK mechanism described in [11], informally.

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. The size of n is dynamic depending on available buffer space at the receiver and congestion in the network. 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 sequence number of the segment most recently acknowledged by the receiver, there is a window of data numbered $i+1$ to $i+n$ 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, where $k \geq i+1$, it means it has successfully received all segments up to k. Segment k is acknowledged by sending a request for segment $k+1$. In TCP, data that is transmitted is kept on a retransmission buffer until it has been acknowledged. Thus, when k is acknowledged, segments with sequence number less than or equal to k are removed from the retransmission buffer. If $k < n+i$, the sender may retransmit segments $k+1$ to $n+i$ from the retransmission buffer. In TCP the decision to retransmit these segments depends on the receipt of duplicate acknowledgments and on time-outs.

Of particular interest are the mechanisms TCP uses to recover from loss, including algorithms for Fast Retransmit [5]. With fast retransmit, when the source receives i duplicate acknowledgments (e.g., $i = 3$) 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.

There are a set of congestion control mechanisms that the source uses, which effectively reduce the sender's transmission window by half on retransmission of a packet based on duplicate acknowledgments [6], which we do not go into detail here. However, the avoidance of the retransmission timeout at the sender improves throughput significantly.

The limitation of the cumulative acknowledgment strategy is that in 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. When multiple packets in a window are dropped, the sender is constrained by the fast retransmit policy to retransmit one packet per round-trip. The lack of a regular "ack-clock" (the timing derived by the sender based on the arrival of acknowledgments) at the sender disables the fast retransmit algorithm. After retransmitting the first packet in the retransmission buffer, the sender waits for a retransmission timeout to retransmit subsequent "holes" in the receiver's sequence number space. This also triggers the congestion control mechanisms of TCP, dropping the sender's transmission window to 1. The consequence of this entire sequence of events is a substantial reduction in throughput.

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 [11]. The mechanism allows the receiver of data to acknowledge non-contiguous 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. Whether the SACKed flag is on or off, segments are only removed from the retransmission buffer when they have been cumulatively acknowledged.

2.1. An example with cumulative ACK

In Figure 1 we show a simple example that illustrates how the (cumulative) ACK mechanism works with TCP Reno [6]. 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

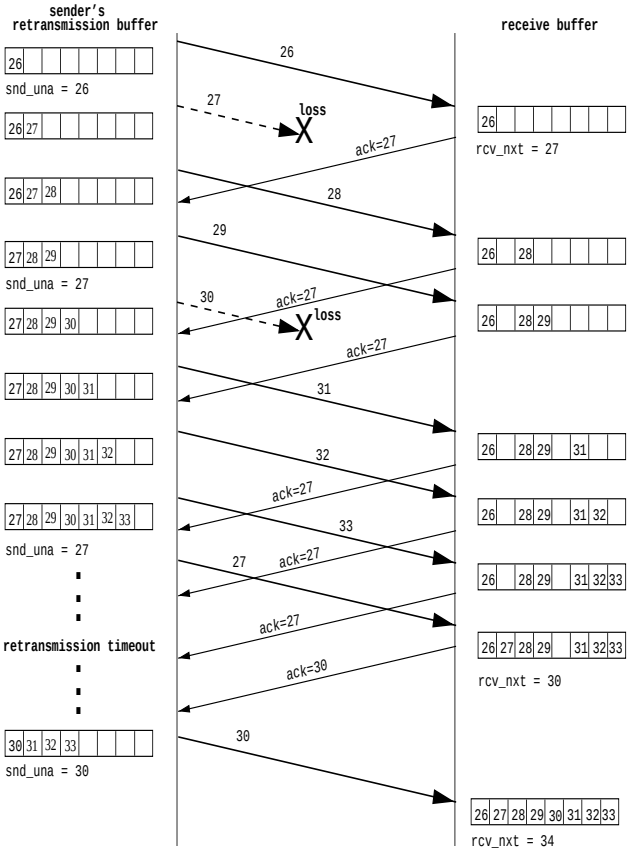


Figure 1: **A simple example illustrating the workings of the ACK mechanism of TCP. The numbers in the figure represents the sequence number associated with a segment of data and is not the actual data.**

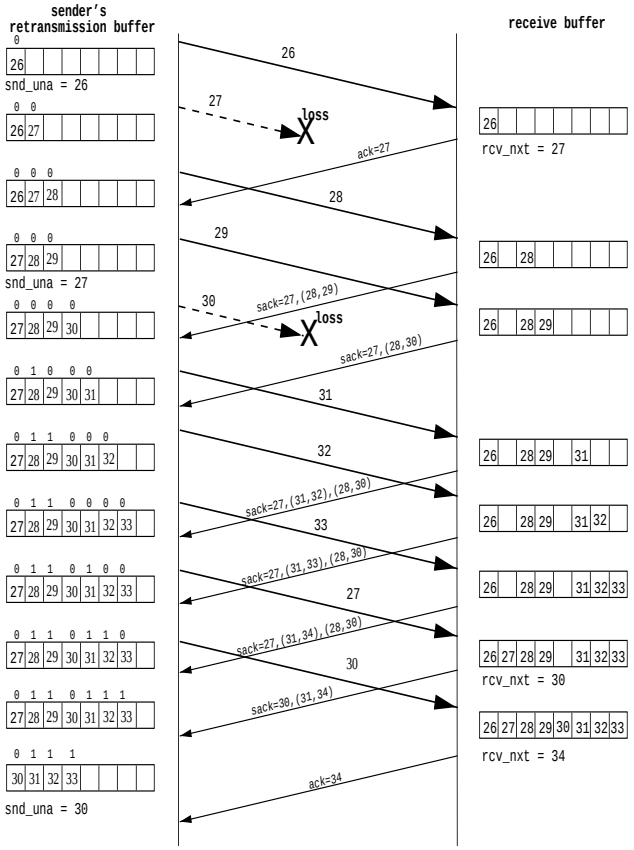


Figure 2: **A simple example illustrating the workings of the SACK mechanism of TCP.**

that the receiver sends back to the sender. The execution illustrated in Figure 1 begins with the sender transmitting segment 26. Then, segment 27 transmitted is lost due to congestion or a bit error. 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 in the receive buffer, 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. 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 to). 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. But, 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 [5].

2.2. An example with SACK

Figure 2 illustrates how the SACK mechanism works on the same set of data as in Figure 1. 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 (S)ACK 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 (the SACK = 27, (31,32),(28,30) indicates that segment 27, and segment 30 were not received). 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.

There is the further question of whether the source's congestion control algorithms are triggered or not when both segment 27 and 30 are lost in the same window. This is dependent on the policy adopted at the source. From the example it is clear that since the source does not go through a timeout, the window does not have to reduce down to 1. It may be adequate for the source to reduce the congestion window by half, as per the congestion control algorithms of TCP Reno [6].

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 time out 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 all the way down to 1.

If multiple packets are lost consecutively after the 1st packet (congestion-related losses are often bursty), then SACK can recover them all even in the most conservative approach. It has the information (say when packets 27, 28 and 29 are lost) with the duplicate SACKs to reliably know that packets 27, 28 and 29 have to be retransmitted. With the cumulative ACK policy, there is no more information as to whether just one packet, 27, was lost or more than that was lost. This is *even* if the most conservative approach (as defined in the RFC) is taken for the retransmission policy at the sender. This scenario is in fact the worst case for the cumulative ACK mechanism, and in Section 5 we analyze this situation to show the improved performance possible with SACK.

3. Formal Specification of the problem

The safety properties we want to show for TCP with with the SACK mechanism is that, data from the sender is not lost, duplicated, or received out of order. In this section we present a formal specification for this problem. Before we present the specification we give a brief description of the formal model we use.

3.1. The I/O Automaton model

The formal model we use to describe the acknowledgment mechanisms is based on the I/O automaton model of of [9]. The automaton consists of four components, a set *states(A)* of states, a nonempty set *start(A)* $\subseteq$ of *states(A)* of start states, a set *acts(A)* of actions, and a set *steps(A)* $\subseteq$*states(A)* $\times$ *acts(A)* $\times$ *states(A)* of steps. The set *acts(A)* can be partitioned into three disjoint sets, *in(A)*, *out(A)*, and *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.

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.

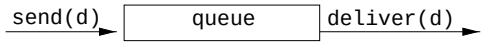
To show that an automaton A "implements" another automaton B we show a *trace* inclusion relationship between them. The set of traces of an automaton consists of the set of sequences of visible actions that the automaton can perform.

3.2. The problem

For the formal specification of the problem we assume a very simple user interface – there is an input action from the user on the sender side to send data message `send(d)`, and on the receiver side data is passed to the user with the `deliver(d)` action.

We define a simple automaton that is an abstract representation of the safety properties we want to show the SACK mechanism satisfies. The basic structure and user interface of the specification is illustrated in Figure 3, and the automaton for the specification, which we call `ReliableQ` is also shown in Figure 3. We describe the automaton in the style of the IOA language [3] for describing I/O automata. In the IOA language an automaton is described by first giving its name followed by the four components of the model mentioned above. Transitions are written in a *precondition, effect* fashion. That is, the state in which an action is enabled is given as a precondition, and the resulting state is given by the effects of the action.

The only state variable of the automaton is `queue` which has type `Seq[Data]` and is initially empty. The type `Seq[Data]` is an ordered list or sequence with elements of type `Data`. We denote the empty sequence as {}. The input action `send(d)` from the user causes data `d` to be added to the tail of the queue. The symbol || is the concatenation operator for sequences in the IOA language. The output action `deliver(d)` 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 `minus(queue, d)` operation removes the elements `d` from the queue. The `prefixes` and `minus` operator are formally defined in full




```
automaton ReliableQ
signature
   input send(m: Seq[Data])
   output deliver(m: Seq[Data])
states
   queue: Seq[Data] := {}
transitions
   input send(d)
   eff queue := queue || d

   output deliver(d)
   pre queue ≠ {}
       d ∈ prefixes(queue)
   eff queue := minus(queue, d)
```

Figure 3: **The top figure shows the basic structure of the specification and the user interface actions, and the bottom figure is the automaton, ReliableQ, for the specification of the reliable message delivery problem.**

paper [14]. 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.

4. The formal model for SACK

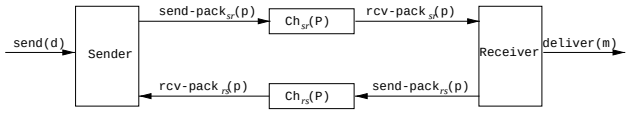


Figure 4: **The structure and the four basic components of the formal model for SACK.**

TCP has the basic structure shown in Figure 4. That is, 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 is assumes the channels, so we need to model them. We model each component as an automaton, and the complete system is the parallel composition of the four component automata. The channels in our model can lose, duplicate, and reorder packets, but they do not corrupt or create spurious packets. They are also parameterized by the type of packets, P, that they transmit. The I/O automaton model for these types of channels are straightforward, and examples of similar channels can be found in [15, 8, 13]. Therefore, we do not include the formal models for the channels here.

4.1. The Sender Automaton

In this section we present a formal I/O automaton model for the sender protocol of the SACK mechanism. The automaton, Sender, is shown in Figure 5. We first specify the type definitions needed to describe some components of the automaton. The DataInt type is the set of pairs that has a unit of data as the first element, and an integer as the second element. The type SackData is the set of triples formed by unit of data, a 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.

```
type DataInt = tuple of Msg: Data, Seqn: Int
type SackData = tuple of Msg: Data, Seqn: Int, Flag: Bool
type Blk = tuple of Left: Int, Right: Int
automaton Sender
signature
   input send(m: Seq[Data]), rcv-pack_rs(seg_ack: Int),
         rcv-pack_rs(seg_ack: Int, b1: Blk, b2: Blk, b3:Blk)
   internal prepare-new-seg(s: Seq[Data]), reset-sack
        prepare-retran-seg(s: Seq[SackData])
   output send-pack_sr(seg: Seq[DataInt])
states
   send_buf: Seq[Data] := {}
   retran_buf: Seq[SackData]
   segment: Seq[DataInt] := {}
   snd_una, snd_nxt: Int := 0
   ready_to_send: Bool := F
transitions

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

   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 := minus(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-pack_sr(seg)
   pre ready_to_send
       seg = segment
   eff ready_to_send := f

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

   input rcv-pack_rs(seg_ack, b1, b2, b3)
   eff if snd_una < seg_ack ≤ snd_nxt then
         retran_buf := delete(retran_buf, seg_ack)
         retran_buf := set_sack(retran_buf, b1, b2, b3)
         snd_una := seg_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 := t

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

Figure 5: **The SACK Sender automaton.**

The states of Sender includes send_buf which is a sequence of data. It holds data received from the user and is initially empty. The retransmission buffer, retran_buf, holds data that may need 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. This buffer is also 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 window is open.

The first transition action shown on the sender side is the `send(d)` input action. This is the action by the external user that passes data to the sender. When the sender receives this input it simply concatenates the data to its send buffer.

The next action is `prepare-new-seg(s)`. This internal action prepares a new segment to be sent. It is only enabled if 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. The available window size is the portion of the window size that is available for new data transmission. It is determined by subtracting `snd_nxt` from the sum of `snd_una` and WS (we assume a constant window size of WS). This action non-deterministically 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 |`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. Next `snd_nxt` is updated, and `ready_to_send` is set to true. This step uses the operators `prefixes`, `init_flag`, `enum`, and `minus`, which are formally defined in the full paper.

The `send-pack`$_{sr}$`(seg)` action places a segment on the outgoing channel, `Ch`$_{sr}$`(P)`, of the sender. This action is enabled if `ready_to_send` is true, and the effect is to set `ready_to_send` to false.

When the sender receives a simple ACK packet, `rcv-pack`$_{rs}$`(seg_ack)`, it checks to see that `seg_ack` acknowledges data that it sent, `snd_una` $<$ `seg_ack` $\leq$ `snd_nxt`. If the condition is true, the acknowledged data is removed from the retransmission buffer, and `snd_una` is updated. When a SACK packet is received, `rcv-pack`$_{rs}$`(seg_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)`. The `set_sack` operator is defined in the full paper. As with the non-SACK acknowledgment, the sender also checks that `seg_ack` acknowledges sent data and removes any acknowledged data from the retransmission buffer.

In TCP-Reno a segment is retransmitted if the retransmission time out expires, or a certain number of duplicate acknowledgments are received (usually three). In our model of the sender, we simplify the mechanism by allowing the sender to generate retransmission segments non-deterministically. Since the sender can non-deterministically 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 the retransmission segments is `prepare-retran-seg(s)`. This action is enabled if no segment is currently being sent, and the retransmission buffer is not empty. 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 non-deterministically chosen from the set. Again the non-determinism in our model allows us to accommodate any implementation policy that determines which hole(s) to fill. The `holes` operator is defined in the full paper.

In the RFC describing for SACK [11], whenever a retransmission timeout expires all the SACK flags in the retransmission buffer are reset to false. In our model we again use non-determinism for a simpler, but more general model. We allow the resetting of the flags, non-deterministically, at anytime. The internal action `reset-sack` resets the sack flags.

4.2. The Receiver Automaton

```
type Blk = tuple of Left: Int, Right: Int
type DataInt = tuple of Msg: Data, Seqn: Int
automaton Receiver
signature
   input  rcv-pack_sr(seg: Seq[DataInt])
   internal drop(s: Seq[DataInt])
   output deliver(m: Seq[Data]), send-pack_rs(seg_ack: Int),
       send-pack_rs(seg_ack: Int, b1: Blk, b2: Blk, b3: Blk)
states
   rcv_buf: Seq[DataInt] := {}
   rcv_nxt: Int := 0
   send_ack, sack_opt: Bool := F
transitions

   input rcv-pack_sr(seg)
   eff send_ack := t
       if rcvnxt ≤ last(seq(seg)) then
         rcv_buf := inst(rcv_buf, delete(seg, rcv_nxt))
         if head(seq(seg)) ≤ rcv_nxt then
           rcv_nxt := seq_max_con(rcv_buf) + 1
         fi
         if rcv_nxt ≤ last(seq(rcv_buf)) then
         sack_opt := t
         else sack_opt := F
         fi
       fi

   output send-pack_rs(seg_ack)
   pre send_ack ∧ ¬ sack_opt
       seg_ack = rcv_nxt
   eff send_ack := F

   output send-pack_rs(seg_ack,  b1, b2, b3)
   pre send_ack ∧ sack_opt
       seg_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 := F

   output deliver(d)
   pre rcv_buf ≠ {}
       d ∈  prefixes(get_data(rcv_buf))
       last(seq(m)) < rcv_nxt
   eff rcv_buf := minus(rcv_buf,  d)

   internal drop(s)
   pre s ∉ cprefixes(rcv_buf)
       s ⊂ rcv_buf
   eff rcv_buf := minus(rcv_buf, s)
```

Figure 6: **The automaton of the SACK Receiver.**

In this section we present the automaton model for the receiver protocol of SACK. The automaton is shown in Figure 6.

The rcv_buf variable of Receiver 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 an acknowledgment segment and an acknowledgment segment with SACK blocks respectively.

When a segment is received from the channel, $\texttt{rcv-pack}_{sr}\texttt{(seg)}$, the receiver sets the send_ack flag to true to enable the sending of an acknowledgment segment. The received segment has new data if rcv_nxt $\leq$ last(seq(seg)). That is, 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. The delete(seg, rcv_nxt) operation returns the part of the segment that has sequence number greater than or equal to rcv_nxt. It is formally defined in the full paper, as is the inst operator. 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 the sequence number of the first byte of data received is less than or equal to rcv_nxt, then rcv_nxt must be 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, seq_max_con(rcv_buf), and adding one to that number. A contiguous prefix is a prefix where the sequence numbers of the elements are contiguous. The seq_max_con operator is defined in the full paper. The receiver must also determine whether to send a regular acknowledgment or a selective acknowledgment. A selective acknowledgment is sent if the receiver has non-contiguous data queued in the receive buffer. That is, if rcv_nxt does not acknowledge the highest sequence number in the receive buffer (rcv_nxt $\leq$ last(seq(rcv_buf))), then sack_opt is set to true.

If send_ack is true and sack_opt is false, then the $\texttt{send-pack}_{rs}\texttt{(seg\_ack)}$ action is enabled. This action sends an acknowledgment where seg_ack is the acknowledgment number. The setting of sack_opt to true enables the $\texttt{send-pack}_{rs}\texttt{(seg\_ack, b1, b2, b3)}$ action, where b1, b2, and b3 are three non-deterministically 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 b2 are different. If there are at least 3, 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 [11], there are also precise rules for determining which blocks should be included in the acknowledgment segment and in what order. A non-deterministic 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 deliver(d). This action is enabled if the receive buffer is not empty, d is a prefix of the data part of the receive buffer, and d does not go beyond the contiguous data that has been received. The get_data operation is formally defined in the full paper. It takes a list with data items sequence number pairs, and returns a list with just the data elements.

The internal action drop(s) results in a non-deterministically chosen sequence of data being dropped from rcv_buf. We include this action because in the proposal for the SACK mechanism [11], a receiver is allowed to drop SACKed data if it receives contiguous data that it does not have room for. The non-deterministic drop(s) action is a generalization of this possibility.

4.3. Complete specification of SACK

An automaton for the SACK 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 just integers or packets that are a tuple of an integer and three SACK blocks. We define APac to be this type. That is,

$$\textbf{type APac} = \texttt{Int} \cup \textbf{tuple of Ack} : \texttt{Int}, \texttt{B1} : \texttt{Blk}, \texttt{B2} : \texttt{Blk}, \texttt{B3} : \texttt{Blk}.$$

Therefore, the composed automaton for SACK is,

$$\texttt{Sack} \triangleq \texttt{Sender}||\texttt{Receiver}||\texttt{Ch}_{sr}\texttt{(P : Seq[DataInt])}||\texttt{Ch}_{rs}\texttt{(P : APac)}$$

5 Performance analysis

Before we prove the safety of SACK, we provide a formal analysis of the improved performance of TCP SACK versus TCP-Reno, when more than one packet is lost from a window of data. Our analysis is for the worst case, which is when the lost packets are the first packets sent in the window.

To model a sender and a receiver that use time we use the *General Timed Automaton (GTA)* model of [8] which is an extension of the I/O Automaton model. In order to do the performance comparison, we also need a formal model of the ACK mechanism of TCP-Reno. We do not include the models here due to space limitations. In our modeling we make the following assumptions: The timing between the sending of packets at the sender is ϵ. The time between the receiving of a segment at the receiver, and the time when an ack is generated for that segment is negligible. The round-trip time is RTT, and packets from the sender take $RTT/2$ to arrive at the receiver, and acknowledgment packets from the receiver take $RTT/2$ to arrive at the sender. Round-trip time is much greater than the time between the sending of segments, and the retransmission timeout is significantly greater than the round-trip time. Also, the time between when the sender receives an acknowledgment that causes it to send and retransmission, and the time when that retransmission is sent is negligible.

In the real protocols round-trip time is variable, as is the time between the sending of segments. However, the variability of these times does not affect our analysis significantly. Given these timing assumptions, we can calculate the worst case latency of packets in a window of data for TCP-Reno and TCP SACK. 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 retransmission strategy suggest by Hoe in [4]. 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 is immediately retransmitted. TCP-Reno actually times out before the second packet is retransmitted. Since the retransmission timeout period is much greater that the round-trip time, the performance of TCP-Reno is actually much worse in this situation than what we show.

Assume the sender has a window of data of size N in its send buffer that it starts sending at time 0. Assume that the first $k \geq 1$ packets get lost. We have the following two theorems, which are stated without proofs.

Theorem 1 *In an execution of TCP-Reno where the first k packets are lost in a data window, the receiver cannot deliver the ith packet in the window before time, $k + 3(\epsilon) + RTT + \min(i-1, k) \times RTT + RTT/2$.*

Theorem 2 *In an execution of TCP SACK where the first k packets are lost in a data window, the receiver may deliver the ith packet in the window at time, $k+3(\epsilon)+RTT+\min(i-1, k) \times \epsilon + RTT/2$.*

Since the inter-packet transmission time is much smaller than the round-trip time, the theorems 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 the number of packets dropped times the round-trip time. This latency results in idleness on the channel for the TCP connection and a consequent loss in throughput.

The latency with the cumulative ACK mechanism is poor in comparison to SACK when the receiver observes all the losses before it generates the third duplicate ack. The receiver, with SACK, observes all k losses before it generates the third duplicate ack when k packets are lost in a burst and three subsequent packets are received. These first k packets can be recovered earlier by SACK. The improvement over the cumulative ACK mechanism is proportional to k times the round-trip time. If all the dropped packets are not in a burst before the three packets are successfully received, then the reduction in latency due to SACK varies according to which packets are dropped and how fast packets are sent relative to the round-trip time. For $k = 1$ the performance of both mechanisms is the same.

6. Proof of safety

For the proof of safety we use the formalization of simulation techniques developed by Lynch and Vaandrager in [10]. Let A be an automaton representing an implementation of a protocol and B be an automaton representing an abstract specification of the protocol. If A and B have the same input and output actions, then a simulation from A to B is relation between states of A and states of B such that certain conditions hold. 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 specification. That is, for each step in the implementation, there must exist a sequence of steps in the specification between states related by the simulation relation to the pre and post-state of the implementation step, such that the sequence of specification steps contains exactly the same external actions as the implementation step, which 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 form A to B, and $A \leq_R B$ via r if r is such a mapping.

6.1 Invariants for SACK

We formally verify that the SACK protocol satisfy 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`. Invariants are properties that are true of all reachable states. These invariants limit the states we need to consider during the simulation. We state the invariants here without proofs. The invariants capture some of the key insights as to why the protocol is reliable.

The first invariant shows the relationships between `snd_una`, `snd_nxt`, and the sequence numbers of the first and last elements of `retran_buf`. The second invariant states that the elements of `retran_buf`, of segments from the sender, and of `rcv_buf`, are sorted in by sequence number. Additionally, the sequence numbers of `retran_buf` and of `segment` are contiguous. The third invariant states that `snd_nxt` is greater than the sequence number of any data in `retran_buf`, `rcv_buf`, or on $\mathtt{Ch}_{sr}$, and that it is greater than or equal to `rcv_nxt`. It also states that when `rcv_buf` is not empty, `rcv_nxt` is one plus the sequence number of the last contiguous piece of data in `rcv_buf`. The final invariant says that elements in buffers or in segments that have the same sequence number part also have the same data part. The variables $\mathtt{in\_transit}_{sr}$ and $\mathtt{in\_transit}_{rs}$ are multi-sets that holds packets from the sender to the receiver and from the receiver to the sender respectively.

Invariant 1

1. If `retran_buf` $\neq \{\}$ then `snd_una` = `head(seq(retran_buf))`.
2. If `retran_buf` $\neq \{\}$ then `snd_nxt` = `last(seq(retran_buf))`$+1$.
3. If `retran_buf` $= \{\}$ then `snd_una` = `snd_nxt`.

Invariant 2

1. If `retran_buf` $\neq \{\}$ then for every element $(d_i, s_i, b_i) \in$ `retran_buf`, except `last(retran_buf)`, $s_{i+1} = s_i + 1$.
2. If `segment` $\neq \{\}$ then for every element $(d_i, s_i) \in$ `segment`, except `last(segment)`, $s_{i+1} = s_i + 1$.
3. If $\mathtt{in\_transit}_{sr} \neq \{\}$ then for every $p \in \mathtt{in\_transit}_{sr}$ and any $(d_i, s_i) \in p$ except `last`(p) $s_{i+1} = s_i + 1$.
4. If $(d_i, s_i) \in$ `rcv_buf` $\wedge$ $(d_j, s_j) \in$ `rcv_buf` $\wedge\ i < j$ then $s_i < s_j$.

Invariant 3

1. `snd_nxt` $\geq$ `rcv_nxt` $\wedge$ `snd_nxt` $>$ `last(seq(segment))` $\wedge$ `snd_nxt` $>$ `last(seq(rcv_buf))` $\wedge\ (\forall p \in \mathtt{in\_transit}_{sr} :$ `snd_nxt` $>$ `last(seq(`p`))`$) \wedge (\forall q \in \mathtt{in\_transit}_{rs} :$ `snd_nxt` $\geq q)$.
2. If `rcv_buf` $\neq \{\}$ then `rcv_nxt` = `seq_max_con(rcv_buf)` $+ 1$.
3. For any $p \in \mathtt{in\_transit}_{rs}$, `rcv_nxt` is greater than or equal to the acknowledgment number of packet p.

Invariant 4

If $(d_h, s_h, b_h) \in$ `retran_buf` and $(d_i, s_i) \in$ `segment` and$(d_j, s_j) \in p$ for any $p \in \mathtt{in\_transit}_{sr}$ and $(d_k, s_k) \in$ `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 $s_j = s_k \Rightarrow d_j = d_k$.

6.2. The refinement mapping

We define a function from *states*(Sack) to *states*(ReliableQ). It is formally defined below.

Refinement mapping R_{rs}
If $s \in$ *states*(Sack) then define $R_{RS}(s)$ to be the state $u \in$ *states*(ReliableQ) such that: u.queue = concatenation of:
- max(cprefixes(get_data(s.rcv_buf)))
- delete(get_data(s.retran_buf), s.rcv_nxt)
- s.send_buf □

It is clear that send_buf in Sack should map to a suffix of the abstract queue, and 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 my 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 rcv_nxt as the demarcation point for data that is included in the mapping from both buffers. In the retransmission buffer, data with sequence numbers greater than or equal to rcv_nxt are included in the mapping, and for the receive buffer, data with sequence numbers less than rcv_nxt are including in the mapping as a prefix of the abstract queue.

6.3. Simulation of steps

In this section we prove that the mapping R_{RS} 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 Sack $\leq_R$ ReliableQ *via* R_{RS}.

Proof: The proof is by induction. For the base case we show that the start states of Sack and ReliableQ are related as defined by R_{RS}. For the inductive case we show that each step of Sack can be simulated by a sequence of steps of ReliableQ with the same trace, such that the mapping defined by R_{RS} is maintained between the pre and post-states of Sack and ReliableQ. The details of the proof are in the full paper. ■

Theorem 3 Sack *safely implements* ReliableQ. *That is, traces(*Sack*)* $\subseteq$ *traces(*ReliableQ*).*

Proof: The theorem follows from the soundness of refinement mappings for proving trace inclusion [10], and from Lemma 1. ■

7. Conclusion

The selective acknowledgment option (SACK) [11] that is being proposed as new standard option for TCP has the potential to improve the recovery mechanisms of TCP when multiple packets are lost from a window of data. This improvement may lead to significantly better throughput in certain cases. Simulation experiments have been done to understand the performance improvement [2] due to SACK. These experiments lend confidence that the protocol operates as expected. However, simulations do not help ensure that the protocol is correct. We believe the SACK mechanism has sufficient added complexly to warrant formal specification and verification of its properties.

In this paper we presented a formal model for the 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 also use our model to prove that in certain worst case scenarios, where there are multiple packets lost consecutively, SACK can reduce latency by a factor proportional to $RTT * (k-1)$, where k is the number of packets lost in a window. Our further work is to understand the consequence of the sender having a more aggressive policy to recover from losses using SACK.

References

[1] M. Abadi and L. Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, 1991.

[2] K. Fall and S. Floyd. Simulation-based comparisons of Tahoe, Reno, and SACK TCP. *Computer Communications Review*, 26(3):5–21, July 1996.

[3] S. J. Garland, N. A. Lynch, and M. Vaziri. IOA: A language for specifying, programming, and validating distributed systems draft. Unpublished manuscript, September 1997.

[4] J. Hoe. Improving the start-up behavior of a congestion control scheme for TCP. In *SIGCOMM Symposium on Communications Architectures and Protocols*, August 1996.

[5] V. Jacobson. Congestion control and avoidance. In *SIGCOMM Symposium on Communications Architectures and Protocols*, pages 314–329, 1988.

[6] V. Jacobson. Modified TCP congestion avoidance algorithm. Email to the end2end-interest Mailing List, 1990.

[7] B. Jonsson. *Compositional Verification of Distributed Systems.* PhD thesis, Uppsala University, 1987. Department of Computer Systems.

[8] N. Lynch. *Distributed Algorithms.* Morgan Kaufmann Publishers, Inc, 1996.

[9] N. Lynch and M. Tuttle. An introduction to input/output automata. *CWI Quarterly*, 3(2), September 1989.

[10] N. Lynch and F. Vaandrager. Forward and backward simulations — Part I: Untimed systems. *Information and Computation*, 121(2):214–233, September 1995.

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

[12] S. Murphy and A. U. Shankar. Connection management for the transport layer: Service specification and protocol verification. Technical Report UMIACS-TR-88-45.1, University of Maryland, June 1988. Revised December. 1989.

[13] M. Smith. *Formal Verification of TCP and T/TCP.* PhD thesis, M.I.T., 1997.

[14] M. A. Smith and K. K. Ramakrishnan. Formal verification of safety and performance properties of TCP selective acknowledgment. Technical report, AT&T Labs Research, October 1998.

[15] J. Søgaard-Anderson, N. Lynch, and B. Lampson. Correctness of communications protocols, a case study. Technical Report MIT/LCS/TR-589, M.I.T., November 1993.