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.

1063-6692/02$17.00 © 2002 IEEE

DOCKET ALARM Find authenticated court documents without watermarks at docketalarm.com.

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

DOCKET ALARM Find authenticated court documents without watermarks at docketalarm.com.

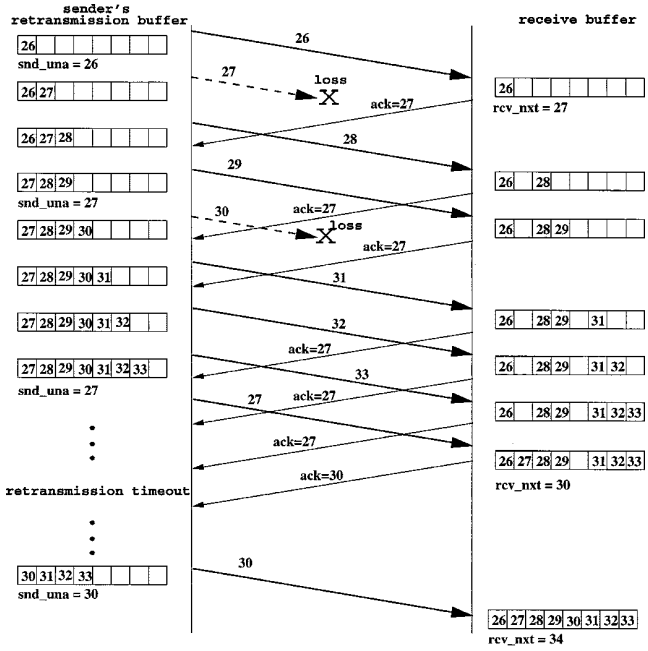

Fig. 1. Example illustrating the workings of the ACK mechanism of TCP.

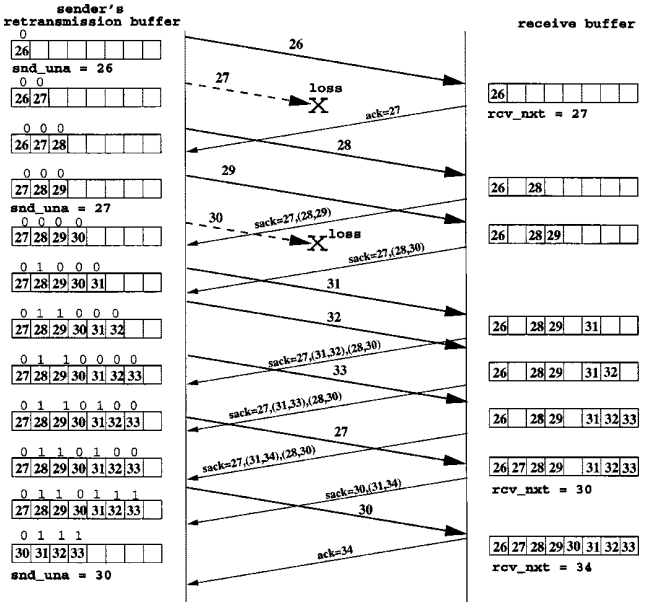

Fig. 2. Example illustrating the workings of the SACK mechanism of TCP.

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.

DOCKET ALARM Find authenticated court documents without watermarks at docketalarm.com.

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

DOCKET ALARM Find authenticated court documents without watermarks at docketalarm.com.

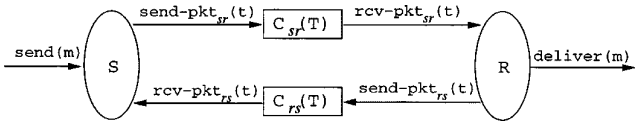

Fig. 4. Structure and the four basic components of the model for SACK.

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, $\mathtt{C_{sr}(T)}$, 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 $\mathtt{send\text{-}pkt_{sr}(t)}$ input action places a packet in the multiset. The complementary output action $\mathtt{rcv\text{-}pkt_{sr}(t)}$ removes the packet from the channel. The internal action $\mathtt{drop_{sr}(t)}$ nondeterministically removes an element from the multiset. The set minus operator for the multiset only removes one copy of an element. The internal action $\mathtt{duplicate_{sr}(t)}$ 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 |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 $\mathtt{send\text{-}pkt_{sr}(seg)}$ action places a segment on the outgoing channel, $\mathtt{C_{sr}(T)}$, of the sender, and $\mathtt{rcv\text{-}pkt_{rs}(ack)}$ takes a simple ACK packet from $\mathtt{C_{rs}(T)}$. First `ack` is checked to see that it acknowledges data that was sent, `snd_una` $<$ `ack` $\leq$ `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, $\mathtt{rcv\text{-}pkt_{rs}(ack, b1, b2, b3)}$, the sender sets the SACK flag of the bytes indicated by the SACK blocks to true with the assignment of `retran_buf` to

DOCKET ALARM Find authenticated court documents without watermarks at docketalarm.com.

DOCKET ALARM

Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

Real-Time Litigation Alerts

Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

Advanced Docket Research

With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

Analytics At Your Fingertips

Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

API

Docket Alarm offers a powerful API (application programming interface) to developers that want to integrate case filings into their apps.

LAW FIRMS
Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

FINANCIAL INSTITUTIONS
Litigation and bankruptcy checks for companies and debtors.

E-DISCOVERY AND LEGAL VENDORS
Sync your system to PACER to automate legal marketing.