Self-Stabilizing Sliding Window ARQ Protocols

John M. Spinelli, *Member, IEEE*

***Abstract*— It is shown that implementing a practical self-stabilizing sliding window protocol requires a bound on the maximum delay or maximum memory of the communication channel involved. This motivates using communication channel models that incorporate a delay or memory bound. For such models, two new ARQ protocols are presented that self-stabilize by using 1 bit of overhead in each transmitted message. The protocols operate like selective repeat ARQ, except that when a fault places them in an incorrect (unsafe) state, the additional bit in the protocol messages allows automatic recovery. Following a transient fault, the bounded delay protocol stabilizes within four round-trip times. The bounded memory protocol stabilizes after sending at most $2(K+n)$ messages, where K the is maximum number of messages that can be stored in one direction on the channel, and n is the window size of the sender.**

***Index Terms*— ARQ protocols, computer network protocols, protocols, self-stabilizing protocols.**

I. Introduction

THE ARQ protocols [4], [5] have long been used in data networks to provide reliable packet delivery over communication channels subject to data loss. They may be used at the data link layer to provide reliable communication over a physical link, or at the transport layer to provide reliable communication between a source and destination. A class of ARQ protocols, called sliding window protocols, are known to operate properly when they are started from a correct (safe) state. However, if some fault, such as memory loss at a processor or some other transient malfunction, causes the protocol to operate from an incorrect (unsafe) state, it has no built-in ability to recover. Incorrect operation can persist for an arbitrarily long time following a transient fault. Typically, incorrect states involve a loss in coordination between the source and the destination. One way to counter coordination loss is to design protocols that are self-stabilizing [11].

A protocol is called self-stabilizing if it is guaranteed to enter a correct state within finite time or after taking a finite number of steps [6]. Therefore, a self-stabilizing ARQ protocol eventually regains coordination and resumes correct operation when started from an arbitrary state. Such a protocol is highly fault tolerant, and can automatically recover from any transient fault.

In this paper, we show that the ability to design self-stabilizing sliding window ARQ protocols depends upon the existence of bounds on the memory or delay of the communication channel used, and that the implementation of existing protocols requires knowledge of these bounds. This motivates the use of delay or memory bounds directly as part of the communication channel model. Two protocols are then developed that exploit the knowledge of a delay or memory bound to self-stabilize. These protocols have finite-state size and add a single bit of overhead to each standard ARQ message. Following a transient fault, the bounded delay protocol stabilizes within $8D$ time units, where D is the one-way channel delay. The bounded memory protocol stabilizes after sending at most $2(K+n)$ messages, where K is the maximum number of messages that can be stored in one direction on the channel and n is the window size of the sender. The protocols are extensions to selective repeat ARQ. With appropriate choice of parameters, they also implement other sliding window ARQ protocols such as go back n and stop and wait.

The remainder of this paper is organized as follows. Section II shows an example of coordination loss and reviews of the methods that have been used to counter it. Section III gives a precise problem definition and describes the channel and protocol model used. Section IV shows that delay or memory bounds are necessary for self-stabilizing ARQ, and presents two self-stabilizing ARQ protocols. A few conclusions follow in Section V.

II. Countering Coordination Loss

To examine the effects of coordination loss arising from a transient fault, we focus on the simplest member of the class of sliding window protocols: stop and wait. Here, and in the rest of this paper, general familiarity with the operation of sliding window protocols such as stop and wait, go back n, and selective repeat is assumed [4], [5].

The correct operation of stop and wait between a source process x and a destination process y is illustrated in Fig. 1. Data packets entering the system at x and leaving the system at y are numbered $p_0, p_1, \cdots$. Process x assigns a sequence number modulo 2 to each data packet, and sends a message containing the packet and its sequence number to process y. These are indicated as $d_0, d_1, \cdots$ in the figure. Similarly, process y sends modulo 2 requests, $r_0, r_1, \cdots$, for subsequent data messages to x that serve as acknowledgments. Since a sequence number, say 0, is reused for every other packet, the source must have a method to determine unambiguously which 0 numbered data packet is being acknowledged. This is accomplished by using the transmission and acknowledgment of the previous packet, with sequence number 1, to sweep the channel clear of old messages. When the source begins to send a new 0 numbered data packet, there are no d_0 or r_1

Manuscript received July 8, 1994; revised July 25, 1996; approved by IEEE/ACM Transactions on Networking Editor I. Cidon. This work was supported in part by a grant from the John D. and Catherine T. MacArthur Foundation.

The author is with the Department of Electrical Engineering and Computer Science, Union College, Schenectady, NY 12308 USA (e-mail: spinelli@grinch.union.edu).

Publisher Item Identifier S 1063-6692(97)02771-4.

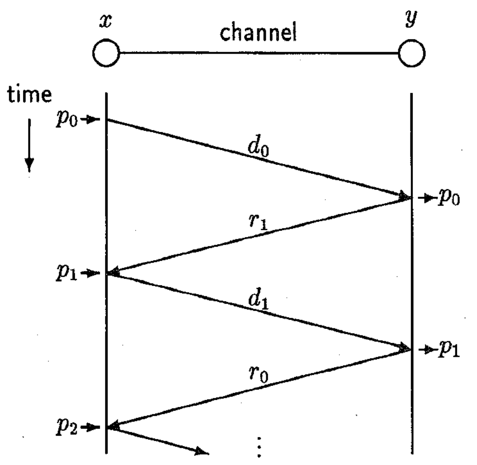

Fig. 1. Correct stop and wait dialog.

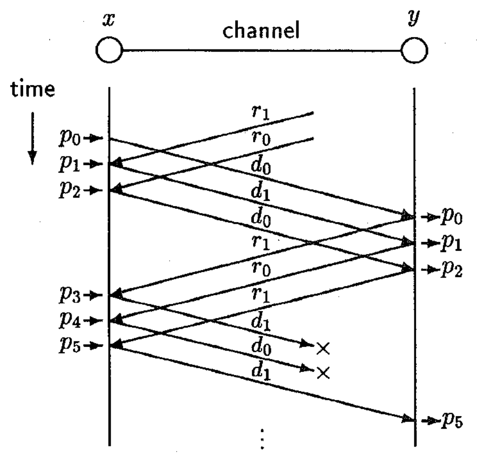

Fig. 2. Incorrect stop and wait dialog.

messages on the channel. This coordination between x and y is maintained in the absence of faults.

The potential for incorrect operation of stop and wait when coordination is lost is illustrated in Fig. 2. Assume that, as a result of a fault, two request messages, r_1 and r_0, are initially present on the channel when the protocol begins. With appropriate message timing, the situation shown in the figure can occur. The channel is in an incorrect state that causes process x to mistake a request for an older packet for a later one. No further faults occur, but the channel may remain in an incorrect state indefinitely. The figure can be modified such that the relative timing of message arrivals is the same as in Fig. 1, so that the processes have no way to detect that an incorrect channel state exists. Should the channel delete one or more data messages at some later time, one or more data packets may be lost. This is indicated at the bottom of Fig. 2. Similar scenarios can be devised that will cause any standard ARQ protocol to lose an arbitrary number of data packets.

A. Previous Work

To examine the behavior of ARQ protocols in a faulty environment, it is assumed that a transient fault leaves the protocol in an arbitrary state, and that this is followed by a sufficiently long fault-free interval during which the protocol tries to recover. We then examine the set of packets that are delivered by the protocol during this interval [3]. It is shown in [8], [9], and [15] that when processes can fail and lose their memory, and there is no bound on the delay or memory of the channel, no protocol can guarantee delivery of every packet following a recovery, even when no further faults occur. In [15], it is shown that delivery can be guaranteed if there is a known upper bound on the transmission delay of the channel.

Several methods have been developed to counter the type of coordination loss indicated in Fig. 2, as well as other types of coordination loss. Many situations under which coordination loss can and cannot be countered are described in [3].

The problem with stop and wait (and other sliding window protocols as well) is that the sequence numbers used form a periodic sequence 0, 1, 0, 1, $\cdots$ that overlaps with a delayed version of itself and allows situations like the one shown in Fig. 2 to go undetected. To avoid this problem, Afek and Brown [1] developed a self-stabilizing ARQ protocol that uses an infinite aperiodic sequence of numbers chosen from a finite alphabet of sequence numbers. Since the sequence of numbers used never repeats, coordination loss cannot persist. This protocol requires infinite memory at the source to generate the infinite aperiodic sequence. If there is a known bound γ on the number of messages that can be stored on the channel, then Afek and Brown point out that using a sequence of numbers whose period is larger than γ can provide self-stability. In Section IV, we present a protocol that extends this idea to selective repeat ARQ. Another approach discussed in [1] and [5] is to use a pseudorandom sequence number to avoid periodicity. Since faults may result in a loss of state information, the ability to generate a random sequence implies access to some external mechanism such as a real-time clock that is not affected by faults.

Gouda and Multari [11] have developed a self-stabilizing sliding window protocol that uses unbounded sequence numbers to avoid periodicity. The protocol allows processes to detect coordination loss and update their sequence numbers to "catch up." An interesting property of their protocol is that it works on non-FIFO channels. Implementing this protocol would require choosing a large, but bounded, sequence number space. In addition, the protocol assumes that processes have the ability to detect an empty channel. In practice, this would most likely be achieved by having a known bound on the maximum transmission delay.

Another approach to self-stabilizing ARQ is to use an ordinary ARQ protocol and periodically run a self-stabilizing initialization protocol that clears the channel of old ARQ packets and restarts the ARQ protocol from a correct initial state. It is shown in [15] that correct initialization can be achieved when there is a known bound on transmission delay, but periodic initialization can introduce considerable inefficiency during ordinary operation. The protocols presented in Section IV effectively perform periodic initialization, but without the need to suspend data transmission during ordinary operation.

III. Problem and Model Description

While a computer network is operating, there may be intervals of time when components behave in a manner that is inconsistent with their operating assumptions. We call such an interval *faulty*. There are certain types of errors, such as data loss on a channel, that are part of the system model and are not considered faults. Two examples of the rare occurrences that can give rise to faulty intervals are described below.

Processor Malfunction: Due to a hardware or software error, a processor could behave unpredictably, or fail completely. This may involve a complete loss of state information.

Channel Malfunction: Some error on a physical channel could fool the error-correcting codes used, and cause a message that was never sent to be received. Messages could also be accidentally duplicated or reordered by an underlying network.

Once the malfunction subsides, a protocol may be left in an uncoordinated state that could never arise through ordinary system operation. Therefore, recovery from a fault can be viewed as the ability of a protocol to eventually operate correctly when started from an arbitrary state. We assume that, following the faulty interval, there is a sufficiently long interval during which no further faults occur, and we examine the ability of a protocol to recover during this interval.

A. Reliable Packet Delivery

We define here the packet delivery problem that ARQ protocols are intended to solve. It is assumed that there is an inexhaustible supply of finite-size data packets $p_0, p_1, \cdots$ to be *read* by the source x, sent to the destination y, and then *written*. For the purpose of analysis, we assume that these packets are numbered sequentially starting from 0. The ARQ protocol does not have access to these packet numbers, and treats data packets as an ordered sequence of black boxes that are to be delivered from x to y without any reordering, duplication, insertion, or deletion. Correct operation of the protocol means that the string of packets read by the source should be the same as the string of packets written by the destination, except that the destination may be behind due to transmission delay or protocol handshaking delay. Therefore, the string of packets written should be a prefix of the string of packets read, and every packet should eventually be written. This criterion may be expressed precisely using the following definitions.

$R(t)$	string of packets read by x up to time t;
$W(t)$	string of packets written by y up to time t.

Reliable Packet Delivery: Delivery of packets from x to y is reliable iff

(safety)	$W(t)$ is a prefix of $R(t)$ for all t and
(liveness)	for each packet p_i there exists a time t_i such that $p_i \in W(t_i)$.

Sliding window ARQ protocols are known to provide reliable packet delivery when they are started from a correct initial state. This is shown informally in [5], and formally for a related protocol in [2], [12], and [10].

If an ARQ protocol is self-stabilizing, then reliable packet delivery should resume some time after the conclusion of a faulty interval. We express the idea of resuming reliable packet delivery by ignoring those packets that entered the system up to some time, and applying the above definition beginning with some data packet p_n. The following definitions express this idea precisely.

$R_n(t)$	string of packets read by x up to time t starting from p_n;
$W_n(t)$	string of packets written by y up to time t starting from the first occurrence of p_n.

Before packet p_n is read, $R_n(t) = \epsilon$, and before packet p_n is written for the first time,[1] $W_n(t) = \epsilon$.

Stabilizing Packet Delivery: Delivery of packets from x to y is stabilizing iff there exists an n and a packet p_n read at time t_n such that

(safety)	$W_n(t)$ is a prefix of $R_n(t)$ for all $t \geq t_n$, and
(liveness)	for each packet $p_i, i \geq n$, there exists a time $t_i \geq t_n$ such that $p_i \in W_n(t_i)$.

This definition depends only on properties of the set of packets read and written by a protocol, and is independent of the state of a protocol following a fault. We are concerned with the design of protocols that provide stabilizing packet delivery regardless of their initial state.

B. Process Model

An ARQ protocol consists of a source process x and a destination process y that can communicate by exchanging *messages* on an asynchronous communications channel. Each process has access to finite local storage, and may also use a finite number of *timers*. A timer is a local variable that can be set to some number of time units, and then counts down toward zero. When a timer reaches zero, it triggers a *time-out*, and remains at zero until it is reset by some procedure. Typically, ARQ protocols reset timers so that an infinite number of time-outs are caused by a given timer, but they are separated by a minimum interval specified by the reset value. Timers are local to the process, and are not synchronized with other processes or a global time reference. A process consists of a set of procedures, one associated with each time-out and each protocol message type that can be received. When a given time-out occurs or when a message is received, the associated procedure is executed. There is also an initialization procedure that sets initial values for the variables and timers. Procedures may manipulate local storage and send messages on the channel. The source process can *read* and store data packets available from a higher layer, and can include them as part of a message. Successive reads return successive data packets, that is, data packets cannot be reread by the source. The destination can store data packets received in messages and can *write* them to a higher layer. The operation of the protocol is independent of the contents of data packets. The state of a process is defined by a value for each of its variables and timers. The possible values for a variable or timer are included as part of its definition.

C. Channel Models

The bidirectional communication channel between x and y is represented by two strings C_{xy} and C_{yx} that are the sequence of finite-length messages being sent from x to y and from y to x, respectively. When process x receives a message,

[1] An incorrect ARQ protocol may write the packet multiple times.

the *head* message of C_{yx} is removed and made available to the appropriate procedure at x. When process x sends a message, it is added to the *tail* of C_{xy}. Sending and receiving messages are considered atomic operations. Similar interactions occur for process y. The channel preserves the order (FIFO) and correctness of messages, but may lose (delete) a message at any time. Each message has some nonzero probability of being received, and the channel does not insert or duplicate messages.

The channel described above will be called the *unbounded channel* because there is no bound on the number of messages that it can store (memory) or how long it can take to deliver them (delay). We also define two variations on this channel that impose restrictions on memory or delay.

Bounded Delay Channel: Each received message has a transmission delay of less than D time units.

Bounded Memory Channel: There are at most K messages in transmission in each direction on the channel. If a process, say x, sends a message while $|C_{xy}| = K$, then the message is deleted.[2]

Any practical network has bounds on memory and delay, but these may not be accurately known. Having a bound on delay effectively implies a bound on memory since a process can read only a finite number of packets in a finite time. It is possible, though, that one bound may be easier to determine for a given network. Assuming an overly large value for D or K does not affect the correctness of the self-stabilizing protocols presented, but does affect the speed at which they self-stabilize.

D. Protocol Execution

The state of an ARQ protocol is defined as a state for each process, x and y, and a sequence of protocol messages in C_{xy} and C_{yx}. If each process is initialized, and the channel is empty in both directions, then the protocol is said to be in its *initial state*. A *correct state* is any state that can be reached by a protocol started from its initial state.

The time-out mechanism allows a process to behave differently when starting from a given state, depending on the relative timing of messages received from the channel. Therefore, in order to uniquely characterize the behavior of a process, it is necessary to specify the interarrival times of the messages that it receives. A *timed message sequence* is a sequence $w = m_1t_1m_2t_2\cdots m_n$ that specifies both a sequence, $m_1m_2\cdots$ of messages and a sequence $t_1t_2\cdots$ of interarrival times between the messages. We say that a process receives a timed messages sequence $w = m_1t_1m_2t_2\cdots m_n$ beginning in state s if it receives m_1 while in state s, then after t_1 time units receives m_2, etc. If a process receives a timed message sequence w beginning in state s, it will move to some unique state s'. We say that s' follows s over the timed messages sequence w. This notation will be used to establish the impossibility result in Section IV-A.

[2]This avoids deadlocks if the channel is full in both directions and each process executes a *send*.

IV. Self-Stabilizing ARQ Protocols

In this section, we consider the design of self-stabilizing ARQ protocols on the three channels previously defined.

A. The Unbounded Channel

On the unbounded channel, there is no limit on the number of messages present on the channel or on the amount of time that they can take to arrive. Because of this, a process cannot determine if a set of received messages is part of a coordinated dialog with another process, or if the messages were left on the channel following a fault. If there is some limit on the number of data packets that the source process can store, then messages left on the channel following a fault can cause the source to discard an unbounded number of undelivered packets. We say that an ARQ protocol is *finite* if it has finite memory and message size. This is explicitly required by the process model of the previous section. The following theorem motivates designing ARQ protocols for bounded delay or bounded memory channels.

Theorem 1: There does not exist a finite ARQ protocol that provides stabilizing packet delivery on the unbounded channel.

Proof: Assume that the theorem is false. We use the standard technique [7], [8], [11], [15] of putting a particular set of messages on the link in the state that follows a transient fault, and show that this will cause the protocol to fail. Consider a finite window protocol that provides stabilizing packet delivery and, due to the finite memory restriction, can store at most P data packets. The liveness property requires that the source x eventually reads packet p_n for every n. Let w_n be a timed message sequence that takes x from its initial state to some state such that it reads packets $p_0p_1\cdots p_n$. At time t_0, assume that x is in its initial state, and that $C_{yx} = m_n$, where m_n contains the messages in w_n. In the interval (t_0, t_n), let x receive the timed message sequence w_n, and assume that each message sent by x during this interval is deleted by the channel. For $n > P$, at time t_n, there are at least $n - P$ data packets in the sequence $p_0p_1\cdots p_n$ that are not stored by the source, are not present on the channel, and have not been written by the destination.

If the protocol provides stabilizing packet delivery, then there exists an i such that $W_i(t)$ is a prefix of $R_i(t)$ for all t, and $p_i \in W_i(t_i)$ for some t_i. If we choose $n > i + P$, and apply the above argument, then at least $i + 1$ packets in the string $p_0\cdots p_n$ are never delivered. Therefore, there is at least one packet in the string $p_i\cdots p_n$ that is never delivered. This contradicts stabilizing packet delivery. □

B. Bounded Delay Channels

If a protocol knows a bound D on channel delay, then it can use a time-out to regain coordination between a source and a destination. As a simple example, consider stop and wait. Suppose that process x sends no messages in the interval $(t_0, t_0 + 2D)$, and that process y sends messages only in response to receiving messages from process x. Then at time $t_0 + D, C_{xy} = \epsilon$, and at time $t_0 + 2D, C_{yx} = \epsilon$ as well. For stop and wait, any combination of states for x and y together with an empty channel is a correct state, so the protocol has self-stabilized at time t_0+2D. Such an approach is impractical because D may be large, and the time-out would have to be used periodically to clear the channel.

1) Stop and Wait: We begin by showing how the simple stop and wait protocol can be made self-stabilizing, and then proceed to modify this protocol to implement selective repeat ARQ.

The bounded delay stop and wait (BDSW) protocol is an extension to stop and wait that adds a second binary sequence number to each protocol message. The standard stop and wait sequence number sn is incremented (modulo 2) each time a new data packet is read by the source. The secondary sequence number st remains constant for at least $2D$ time units, and is then incremented (modulo 2) when the next data packet is read. The destination process y operates exactly as in stop and wait, except that it returns the received st value with each acknowledgment. The source process operates exactly as in stop and wait, except that it includes its stored st value in each transmitted message, and ignores any acknowledgment that contains a value of st that does not match its stored value. This consistency check on the value of st allows the protocol to regain coordination.

A formal description of the bounded delay protocol is shown in Fig. 3. We use the stop and wait convention where the destination acknowledges receiving a data packet by requesting the next numbered packet (modulo 2). The protocol uses two timers. One, T_r, is the standard stop and wait retransmission timer. It can be reset to any positive value. The other, T_s, is used to schedule a change in st. It is reset to a value $>2D$. The variable *change* is used as a flag indicating that at least $2D$ time units have expired since st was last changed. It is important that st be changed only when a new data packet is read. This avoids transmitting a given data packet with two different values of st. A proof of correctness for BDSW can be found in [14].

2) Formal Description of BDSW: In the code that follows, the scope of conditionals and loops is implied by indentation, and comments begin with //.

```
SourceProcess: $x$
Variables:
    $sn$: $\{0,1\}$
    $st$: $\{0,1\}$
    $change$: {yes, no}
    $M$: {data packet}
Timers:
    $T_r$: $\{\geq 0$, reset to a value $> 0\}$
    $T_s$: $\{\geq 0$, reset to a value $> 2D\}$
Procedures:
    0. Initialization:
        $sn := 0; st := 0;$ $change$ := no;
        set $T_r$; set $T_s$;
        $M$ := read_packet;
        send $DATA(sn, st, M)$;
    1. Time-out $T_r$:
        set $T_r$;
        send $DATA(sn, st, M)$;
    2. Time-out $T_s$:
        $change$ := yes;
    3. Receive $REQ(i, j)$:
        if $sn \neq i$ and $st = j$ then
            // this is a valid acknowledgment.
            if $change$ = yes then
                //$st$ can be changed since a new
                // packet is being read in.
                $change$ := no; set $T_s$;
                $st := (st + 1)$ mod 2;
            $sn := (sn + 1)$ mod 2;
            set $T_r$;
            $M$ := read_packet;
            send $DATA(sn, st, M)$;
DestinationProcess: $y$
Variable:   $rn$: $\{0,1\}$
Procedures:
    0. Initialization:    $rn := 0$;
    1. Receive $DATA(k, l, P)$:
        if $rn = k$ then
            write_packet($P$);
            $rn := (rn + 1)$ mod 2;
        send $REQ(rn, l)$;
```

3) Selective Repeat: The technique of BDSW can be used to achieve self-stabilizing selective repeat ARQ, but problems caused by implicit acknowledgments and out-of-bounds acknowledgments must be addressed. We now consider a selective repeat ARQ protocol that has a window size of n at the source and destination, and a modulus of $m \geq 2n$ used for the sequence number space.

In go-back-n or selective repeat ARQ, a given packet i in the source window may be explicitly acknowledged by receiving a request for packet $i+1$, or implicitly acknowledged by receiving a request for a later packet. This creates a difficulty for the consistency check on the secondary sequence number st, since it may have changed at the source after the source sent packet i, but before packet i was implicitly acknowledged. This problem is avoided by performing the consistency check on st only on packets that have a sequence number $sn = 0$. Accordingly, st is allowed to change only when the next $sn = 0$ numbered packet is read. Data messages with $sn \neq 0$ carry the st value associated with the last $sn = 0$ packet read in since they may cause the destination to send an implicit acknowledgment of the last $sn = 0$ packet.

An additional problem unique to selective repeat is the possibility that out-of-bounds acknowledgments may remain on the channel following a fault. In normal operation of selective repeat, each acknowledgment received by the source requests either the next packet that the source can read or one of the packets stored in the source window. When an acknowledgment outside this range is received by the source, it indicates that a fault has occurred. In this situation, we clear the channel using the method described at the start of Section IV-B, and reinitialize the protocol.

We present a bounded delay selective repeat (BDSR) below that applies the technique of BDSW to selective repeat (SR) ARQ, with the two modifications mentioned above. BDSR is an extension to SR that adds a second binary sequence number to each protocol message. The standard SR sequence number sn is incremented (modulo m) each time a new data packet is read by the source. The secondary sequence number st remains constant for at least $2D$ time units, and is then incremented (modulo 2) when the next $sn = 0$ data packet is read. The destination process y operates like SR, except that it returns the most recently received st value with each acknowledgment. The source process operates like SR, except that it includes its stored st value in each transmitted message, and performs a consistency check on each acknowledgment of an $sn = 0$ numbered data packet. If such an acknowledgment does not contain an st value that matches the currently stored value at the source, then the acknowledgment is ignored. This consistency check on the value of st prevents SR from accepting old acknowledgments left on the channel as a result of a transient fault, and allows the protocol to regain coordination.

A formal description of BDSR is given in Section IV-B4. BDSR imposes the same restriction on m as SR, namely, $m \geq 2n$. The protocol sends $DATA, REQ$, and $RESET$ messages. We use the SR convention where a request (REQ message) by the destination for the next numbered packet implicitly acknowledges all previous packets in the source window. Selective repeat acknowledgments may also include selective acknowledgment of later packets in the source window. BDSR allows up to $n - 1$ additional acknowledgment bits in a REQ message so that any packet in the source window may be acknowledged. Although the selective acknowledgment function is important for efficient operation of SR and BDSR, correctness does not depend upon it. The correct operation of BDSR is independent of the use of selective acknowledgments by the destination and of the retransmission strategy employed by the source. The only requirement is that the first packet in the source window is transmitted periodically. In the presentation that follows, cyclic retransmission of unacknowledged packets in the source window is assumed for simplicity.

The format for data messages is $DATA(sn, st, P)$, where sn and st are the primary and secondary sequence numbers at the source, and P is a data packet. The format for acknowledgment messages is $REQ(rn, rst, B)$ where rn and rst are the primary and secondary sequence numbers at the destination, and B is an $n - 1$ bit string of acknowledgments discussed above. The source maintains an array M of data packets in the source window. The window begins with sn and ends at $(sn + n - 1) \mathrm{mod} m$. A second array A indicates which packets in the source window have been acknowledged. Similarly, the destination maintains an array M of data packets in the destination window. This window runs from rn to $(rn + n - 1) \mathrm{mod} m$. A second array A indicates which packets in the destination window await being written, and is similarly indexed. For notational convenience, M and A are indexed over a range of $0, \cdots, m - 1$, but only n of these indexes are used by the protocol to store packets or acknowledgment bits at any one time[3].

[3]Readers interested primarily in self-stability may ignore references to array A in BDSR. This array is used to improve efficiency by implementing selective retransmission of packets.

In the absence of faults, all $REQ(rn, rst, B)$ messages have $rn \in \{sn, \cdots, sn + n\}$. Receiving an out-of-bounds REQ message indicates that the source and destination have lost coordination due to a transient fault. In this case, a reset is performed, and the channel is cleared using the method described above. Resets are needed only in the case of selective repeat ARQ. For go-back-n or stop and wait, m may be chosen equal to $n + 1$ so that there are no out-of-bounds acknowledgments.

The protocol uses the same two timers as BDSW.

BDSR may be operated as go-back-n by fixing the value of $A[i] := no$ at the source for all i. This prevents the source from skipping acknowledged packets when retransmitting. BDSR may be operated as stop and wait by setting the window size n to one.

4) Formal Description of BDSR: In the following presentation, all ranges, increments, and addition are modulo m unless otherwise stated.

```
Constants :
  $n$: window size
  $m$: modulus
  $D$: upper bound on channel delay
Source Process : $x$
Variables :
  $reset$: {yes, no}
  $sn$: $\{0, \cdots, m-1\}$
  $st$: $\{0, 1\}$
  $change$: {yes, no}
  $M[i]$: {data packet} for $i = 0, \cdots, m-1$
  $A[i]$: {yes, no} for $i = 0, \cdots, m-1$
Timers :
  $T_r$: {$\geq 0$, reset to a value $> 0$}
  $T_s$: {$\geq 0$, reset to a value $> 2D$}
Procedures :
  0. Initialization:
      $sn := 0; st := 0$
      $reset :=$ no; $change :=$ no
      set $T_r$; set $T_s$
      for $i$ from $sn$ to $sn + n - 1$
         $M[i] :=$ read_packet
         $A[i] :=$ no
         send $DATA(i, st, M[i])$
  1. Time-out $T_r$ :
      set $T_r$
      if $reset =$ no
         for $i$ from $sn + 1$ to $sn + n - 1$
            // resend unack'ed. packets
            if $A[i] = no$
              send $DATA(i, st, M[i])$
         // always send most recent packet
         send $DATA(sn, st, M[sn])$
      else send $RESET$
  2. Time-out $T_s$:
      $change :=$ yes
  3. Receive $REQ(rn, rst, B)$:
      if $reset =$ no
         for $i$ from 1 to $n - 1$
            // record received ack bits
            $A[rn + i] := B[i]$
```

```
        if $rn \in \{sn+1, \cdots, sn+n\}$
          // one or more packets are ack'ed.
          if $0 \in \{sn, \cdots, rn-1\}$ and $st \neq rst$
            // this message tries to ack a 0
            // numbered packet, but $st$ does not
            // match, so ignore the message
          else
            // for each ack'ed. packet
            for $i$ from $sn+n$ to $rn+n-1$
                // change $st$ only if a 0
                // numbered packet is ack'ed.
                if $change$ = yes and $i = 0$
                  $st := (st+1) \bmod 2$
                  $change :=$ no
                  set $T_s$
                // refill the source window
                $M[i] :=$ read_packet
                $A[i] :=$ no
                send $DATA(i, st, M[i])$
                set $T_r$
            // slide window
            $sn := rn$
        if $rn \in \{sn+n+1, \cdots, sn-1\}$
            // out-of-bounds ack
            $reset :=$ yes
            pause $2D$
            send $RESET$
            set $T_r$
    4. Receive $RESET$:
        if $reset$ = yes
          go to Initialization Procedure
```

```
Destination Process : $y$
Variables :
  $rn$: $\{0, \cdots, m-1\}$
  $M[i]$: {data packet} for $i = 0, \cdots, m-1$
  $A[i]$: {yes, no} for $i = 0, \cdots, m-1$
Procedures :
  0. Initialization:
    $rn := 0$
    for $i$ from $rn$ to $rn+n-1$
        $A[i] :=$ no
  1. Receive $DATA(sn, st, P)$:
    if $sn \in \{rn, \cdots, rn+n-1\}$
        // packet is in receive window
        $M[sn] := P$
        $A[sn] :=$ yes
    $old\_rn := rn$
    // check if window can slide
    for $l$ from $old\_rn$ to $old\_rn+n-1$
        if $A[l]$ = yes
          // write packet and slide window
          write_packet $M[l]$
          $A[l] :=$ no
          $rn := l+1$
        else break
    send $REQ(rn, st, A[rn+1], \cdots, A[rn+n-1])$
  2. Receive $RESET$:
    send $RESET$
    go to Initialization Procedure
```

5) Proof of Correctness: We first show that BDSR provides reliable packet delivery when correctly initialized, and then we show that it provides stabilizing packet delivery. To show correctness, we use the fact that SR provides reliable packet delivery when started in its initial state [5], and we relate BDSR to it. A formal statement of SR can be obtained by removing from BDSR all references to st, T_s, and *change*. We begin with two lemmas that state properties of SR that are useful in proving the correctness of BDSR.

Lemma 1: If SR is in a correct state at time t_n, then it provides stabilizing packet delivery beginning with the next packet, read by the source after time t_n.

Proof: Consider a run of SR that starts in its initial state and passes through state s at time t_n. Since s is a correct state, such a run exists. This run provides reliable packet delivery, so it also provides stabilizing packet delivery for each packet read following t_n. Now, consider any run of SR that passes through state s at some time t_n. Assuming a given behavior of the channel for $t > t_n$, this run will be identical to the above run for $t > t_n$. Therefore, it also provides stabilizing packet delivery for each packet read following t_n. □

Theorem 2: When started from a correct state, BDSR provides stabilizing packet delivery.

Proof: We show that when BDSR is in a correct state, the value of st does not affect its operation and no *RESET* messages are sent. Under these conditions, SR and BDSR read and write the same set of packets, so by Lemma 1, BDSR provides stabilizing packet delivery.

(safety) Each $REQ(rn, rst, B)$ message that arrives at x may be ignored by x if $rst \neq st$. Assume that any ignored messages are instead deleted by the channel C_{yx}. Under this assumption, the sequence of *REQ* and *DATA* messages sent by BDSR is identical to the sequence that would be sent by SR, except that each BDSR message contains a value of st. Since SR is safe despite channel deletions, BDSR is also safe provided that it sends no *RESET* messages.

For SR, it is known that each *REQ* message received by x has an rn value in the range $sn, \cdots, sn+n$. This is true of BDSR as well since, by the above argument, it sends the same set of *REQ* messages as SR. Therefore, each *REQ* message received by x will be "in bounds," and no *RESET* messages will be sent.

(liveness) To show liveness, we show that the comparison of st and rst made by x will always be satisfied. Therefore, BDSR will accept the same set of *REQ* messages as SR, and the two protocols are equivalent when started from a correct state.

Let y send a $REQ(rn, rst, B)$ packet at time t_1, and let it arrive at x at time t_2. Assume that at $t_2, 0 \in \{sn, \cdots, rn-1\}$ at x. Therefore, this *REQ* acknowledges a 0 numbered packet at x. Call this packet p. To satisfy stabilizing packet delivery, the last 0 numbered packet written by y before t_1 must be p. Each *DATA* message containing p sent by x also contained st. Therefore, at t_1, y has $st = rst$. The value of st at the source cannot change unless another 0 numbered packet is read in, so $st = rst$ at x at time t_2 as well. This shows that the check on the value of st at x will always be satisfied when BDSR is in a correct state. □

The following theorem shows that BDSR converges to a correct state when started from an arbitrary state.

Theorem 3: BDSR provides stabilizing packet delivery on bounded delay channels.

Proof: Assume that BDSR is in an arbitrary state at time t_0. We will show that it converges to a correct state within a finite time. Once in a correct state, Theorem 2 applies, and stabilizing packet delivery is provided. Two cases are considered below.

Case 1: Assume that x does not send any *RESET* messages for a sufficiently long interval following t_0. This implies that all $REQ(rn, rst, B)$ messages received by x have rn in the range $\{sn, \cdots, sn+n\}$. We first show safety and liveness under this assumption, and then consider separately the case when it does not hold.

(liveness) We show that following t_0, sn at x must be incremented, so that the source continues to read packets. Under the above assumption, x will periodically send $DATA(sn, st, M[sn])$ messages on C_{xy} until packet $M[sn]$ is acknowledged. Eventually, one or more of these messages will be received by y. Process y responds to each $DATA(sn, st, M[sn])$ message with a $REQ(rn, rst, B)$ message, where $rst = st$. If $sn = rn$ at y when the *DATA* message arrives, then y increments rn to $sn+1$ before sending the *REQ* message. If $sn \neq rn$ when the data message arrives, then $rn \in \{sn+1, \cdots, sn+n\}$. Eventually one or more of these *REQ* messages will arrive at x. Assume that sn has not already changed at x when an *REQ* message arrives. Since st can only change when a packet is read in, $st = rst$ still holds at x. Receipt of this *REQ* message will cause sn to be incremented since $rn \in \{sn+1, \cdots, sn+n\}$. This shows that the x can continue to read data packets and BDSR cannot deadlock.

(safety) Let t_1 be a time when an $sn = 0$ numbered packet is read by x and st is changed from 0 to 1. The above liveness proof implies that such a time must exist. Let t_2 be a later time at which this 0 numbered packet is acknowledged[4] by a $REQ(i, rst, B)$ message received on C_{yx}. Let t'_2 be the time that this *REQ* message was sent on C_{yx}. We will show that at time t_2, BDSR is in a correct state.

In the interval $(t_1 - 2D, t_1)$, no *DATA* messages with $st = 1$ were sent by x. Therefore, at time t_1, there are no messages with $st = 1$ on C_{xy} or C_{yx}. Since the $REQ(i, rst, B)$ message received at t_2 acknowledges a packet with $sn = 0$, it must contain $rst = st = 1$, and $i \in \{1, \cdots, n\}$. Therefore, this *REQ* message was sent by y after time t_1 in response to a *DATA* message received on C_{xy} in the interval (t_1, t'_2). Call the time that this data message was received t'_1. Fig. 3 illustrates the relationship between these times: $t_1 < t'_1 < t'_2 < t_2$. At time t_2, there are no $st = 0$ messages on C_{xy} or C_{yx}, and $st = 1$ and $sn = i$ at x.

At time t_2, C_{xy} may contain only those *DATA* messages that were sent by x in the interval (t_1, t_2). These data messages contain packets that may have been in the source window at the same time as packet 0 and have sequence numbers in the range $sn \in \{0 - n + 1, \cdots, 0 + n - 1\}$. There are $2n - 1$ sequence numbers is this range. The range does not include

[4]By "acknowledged," we mean that the packet is removed from the source window.

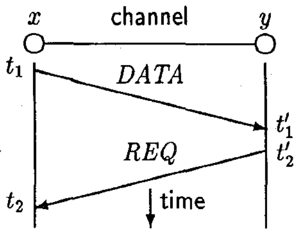

Fig. 3. Timing diagram for proof of Theorem 3.

the sequence number $sn = n$, so there are no *DATA* messages on C_{xy} with $sn = n$ in the interval (t'_1, t_2).

At time t_2, C_{yx} may contain only those *REQ* messages that were sent by y in the interval (t'_2, t_2). At time t'_2, the value of rn at y is $i \in \{1, \cdots, n\}$. In the interval $(t'_2, t_2), y$ may increment rn, but its value may not exceed n because there are no *DATA* messages on C_{xy} with $sn = n$. Therefore, at t_2, C_{yx} contains *REQ* messages with $rn \in \{1, \cdots, n\}$, and the value of rn at y is in the range $\{1, \cdots, n\}$.

The states of x, y, C_{xy}, and C_{yx} described above at time t_2 form a correct state for BDSR. This completes the proof for Case 1.

Case 2: We now show that if x sends a *RESET* message before BDSR can stabilize under Case 1, then following the reset, BDSR is in a correct state. Before x sends a *RESET* message, it pauses for $2D$ time units and ignores all received messages. Following this pause, C_{xy} and C_{yx} are empty. Process x then periodically sends *RESET* messages to y. Eventually, one of these messages reaches y and the reply *RESET* message reaches x. At this point, x and y are in their initial states, and C_{xy} and C_{yx} are empty except for *RESET* messages. Receipt of these *RESET* messages will not change the state of x or y. This is the initial state for BDSR, so it a correct state. □

C. Bounded Memory Channels

If a protocol knows a bound K on the maximum number of messages that can be present in each direction on the channel, then it can use multiple transmissions to clear the channel of old messages and regain coordination. Two ways to do this are to use a large sequence number space, or a long aperiodic sequence of numbers [1], [11]. We show that self-stabilization can be accomplished with a simple protocol that involves only one bit of message overhead as compared to SR.

The bounded memory selective repeat (BMSR) protocol is an extension to SR that adds a second binary sequence number to each protocol message. This secondary sequence number st functions as in BDSR, but a different rule is used to decide when to change its value. A given value of st is used by the source to send at least $2K$ data packets, and is then incremented modulo 2 after at least $2K$ replies are received. The protocol uses only the standard retransmission timer T_r used by SR. The variable *count* is used to keep track of the number of data packets received with a given value of st. A similar mechanism is used in the second protocol of [7].

When an out-of-bounds *REQ* message is received, a reset is performed, and the channel is cleared of old messages. This is accomplished by sending and receiving at least $2K$ *RESET* messages. As with BDSR, resets are not needed for go-back-n or stop and wait.

Other aspects of BMSR, including the entire destination process, are identical to BDSR. A formal statement of BMSR is in the Appendix. The correctness proof for BMSR is similar to the proof of BDSR. The details are omitted.

D. Convergence Rates

Using the proof of BSDR as a guide, upper bounds on the convergence rates of BDSR and BMSR can be established. We consider a fault to end at time t_0, and assume that for a sufficient interval thereafter, no faults occur and no messages are lost by the channel.

1) Convergence Rate of BDSR: Referring to the proof of BDSR, we are interested in the maximum length of the interval from t_0 when the fault ends until t_2 when the protocol has stabilized. Following t_0, it may take up to $2D$ time units for timer T_s to expire and cause *change* = yes. If we assume that $m = 2n$ for selective repeat, then the sender's window may need to move $2n$ packets forward before the next $sn = 0$ numbered packet is read and st changes. This requires at most two round-trip times. Therefore, the length of time interval (t_0, t_1) is at most three round-trip times, or $6D$ time units. At most, one more round-trip time is required for the $sn = 0$ numbered packet to be acknowledged at time t_2. This leads to a convergence time bound of four round-trip times, or $8D$ time units.

For self-stabilizing stop and wait, the above bound can be improved by $2D$ because st may be changed when the next packet is read in, regardless of its sequence number sn. This leads to a convergence time of three round-trip times for stop and wait.

2) Convergence Rate for BMSR: For BMSR, we are interested in determining a bound on the maximum number of messages that process x sends before the protocol stabilizes. Following a fault, $2K$ messages may be sent before *change* = yes. In order to read in and have acknowledged the next $sn = 0$ numbered packet, x may need to move its window forward $m = 2n$ packets. This leads to a maximum of $2K+2n$ messages before the protocol stabilizes.

For self-stabilizing stop and wait on the bounded memory channel, the above bound can be improved slightly because st may be changed when the next packet is read in after *change* = yes. This leads to a maximum of $2K + 1$ messages before the protocol stabilizes. Afek and Brown [1] have developed a self-stabilizing version of the alternating bit protocol that can be operated on a bounded memory channel with a sequence of numbers whose period is larger than $2K$. The amount of memory required to generate this sequence is comparable to that used by BMSR for the variable *count*. They show that a probabilistic version of their protocol sends $O(K)$ messages before converging to a correct state. This convergence rate is similar to BMSR when it is operated as stop and wait.

V. Conclusion

The protocols presented are simple extensions to selective repeat, and impose minimal overhead. In return, they offer considerable fault tolerance. Although the type of faults under consideration in this paper are rare, they have occurred [13], and can be expected to occur on any complex network that operates for a long enough time. As networks become more complex and interconnected, it becomes more impractical to reset them manually following a fault. The automatic recovery provided by self-stabilizing ARQ protocols is an attractive alternative.

Appendix
Formal Description of BMSR

```
Constants :
  $n$: window size
  $m$: modulus
  $K$: upper bound on channel messages
Source Process : $x$
Variables :
  $reset$: {yes, no}
  $sn$: $\{0, \cdots, m-1\}$
  $st$: $\{0, 1\}$
  $change$: {yes, no}
  $count$: $\{0, 1, \cdots, 2K-1\}$
  $M[i]$: {data packet} for $i = 0, \cdots, m-1$
  $A[i]$: {yes, no} for $i = 0, \cdots, m-1$
Timer :
  $T_r$: {$\geq 0$, reset to a value $> 0$}
Procedures :
  0. Initialization:
     $sn := 0; st := 0$; count $:= 0$
     reset $:=$ no; $change :=$ no
     set $T_r$
     for $i$ from $sn$ to $sn + n - 1$
        $M[i] :=$ read_packet
        $A[i] :=$ no
  1. Ti me-out $T_r$ :
        same as BDSR
  2. Receive $REQ(rn, rst, B)$:
     if $reset =$ no
        $count := (count + 1) \bmod (2K)$
        if $count = 0$
          $change :=$ yes
        for $i$ from $1$ to $n - 1$
          $A[rn + i] := B[i]$
        if $rn \in \{sn + 1, \cdots, sn + n\}$
          if $0 \in \{sn, \cdots, rn - 1\}$ and $st \neq rst$
            (ignore message)
          else
            for $i$ from $sn + n$ to $rn + n - 1$
              if $change =$ yes and $i = 0$
                $st := (st + 1) \bmod 2$
                $change :=$ no
                $count := 0$
              $M[i] :=$ read_packet
              $A[i] :=$ no
              send $DATA(i, st, M[i])$
              set $T_r$
          $sn := rn$
        if $rn \in \{sn+n+1, \cdots, sn-1\}$
            $reset :=$ yes
            $count := 0$
            for $i$ from $1$ to $2K$
                send $RESET$
            set $T_r$
    3. Receive $RESET$:
        if $reset =$ yes
            $count := (count+1)$ mod $(2K)$
            if $count = 0$
                go to Initialization Procedure
Destination Process: $y$
    same as BDSR
```

References

[1] Y. Afek and G. M. Brown, "Self-stabilization over unreliable communication media," *Distrib. Computing*, vol. 7, pp. 27–34, 1993.

[2] A. V. Aho, D. Wyner, and M. Yannakakis, "Bounds on the size and transmission rate of communications protocols," *Comput. Math. Appl.*, vol. 8, no. 3, pp. 205–214, 1982.

[3] H. Attiya, S. Dolev, and J. L. Welch, "Connection management without retaining information," *Inform. Computation*, vol. 123, pp. 155–171, Dec. 1995.

[4] A. E. Baratz and A. Segall, "Reliable link initialization procedures," *IEEE Trans. Commun.*, vol. 36, pp. 144–152, Feb. 1988.

[5] D. Bertsekas and R. Gallager, *Data Networks*, 2nd ed. Englewood Cliffs, NJ: Prentice-Hall, 1992.

[6] E. W. Dijkstra, "Self-stabilizing systems in spite of distributed control," *Commun. ACM*, vol. 17, pp. 643–644, Nov. 1974.

[7] S. Dolev, A. Israeli, and S. Moran, "Resource bounds for self-stabilizing message driven protocols," in *Proc. 10th ACM Symp. Principles of Distributed Computing*, Aug. 1991, pp. 281–293; also, *SIAM J. Computing*, to be published.

[8] A. Fekete, N. Lynch, Y. Mansour, and J. Spinelli, "The impossibility of implementing reliable communication in the face of crashes," *J. Assoc. Comput. Mach.*, vol. 40, pp. 1087–1107, Nov. 1993.

[9] ——, "The data link layer: The impossibility of implementation in face of crashes," MIT Lab. for Comput. Sci., Rep. MIT/LCS/TM–355.b, Aug. 1989.

[10] M. G. Gouda, "On 'A simple protocol whose proof isn't,' " *IEEE Trans. Commun.*, vol. COM-33, pp. 380–382, Apr. 1985.

[11] M. G. Gouda and N. J. Multari, "Stabilizing communication protocols," *IEEE Trans. Comput.*, vol. 40, pp. 448–458, Apr. 1991.

[12] B. Hailpern, "A simple protocol whose proof isn't," *IEEE Trans. Commun.*, vol. COM-33, pp. 330–337, Apr. 1985.

[13] E. C. Rosen, "Vulnerabilities of network control protocols: An example," *Comput. Commun. Rev.*, July 1981.

[14] J. M. Spinelli, "Self-stabilizing ARQ on channels with bounded memory or bounded delay," in *Proc. IEEE INFOCOM 1993*, vol. 3, San Francisco, CA, Mar. 1993, pp. 1014–1022.

[15] ——, "Reliable data communication in faulty computer networks," Ph.D. dissertation, MIT, and MIT Lab. for Inform. and Decision Syst. Rep. LIDS-TH-1882, June 1989.

John M. Spinelli (S'81–M'89) received the B.E. degree (summa cum laude) in electrical engineering from The Cooper Union, New York, in 1983, and the S.M. and Ph.D. degrees from the Massachusetts Institute of Technology, Cambridge, in 1985 and 1989, respectively.

He is an Associate Professor in the Department of Electrical Engineering and Computer Science, Union College, Schenectady, NY. His research interests include fault-tolerant protocol design, computer networks, and distributed algorithms.