Block Acknowledgment: Redesigning the Window Protocol

Geoffrey M. Brown, *Member, IEEE,* Mohamed G. Gouda, and Raymond E. Miller, *Fellow, IEEE*

Abstract— **We describe a new version of the window protocol where message sequence numbers are taken from a finite domain and where both message disorder and loss can be tolerated. Most existing window protocols achieve only one of these two goals. Our protocol is based on a new method of acknowledgment, called *block acknowledgment* where each acknowledgment message has two numbers m and n to acknowledge the reception of all data messages with sequence numbers ranging from m to n. Using this method of acknowledgment, the proposed protocol achieves the two goals while maintaining the same data transmission capability of the traditional window protocol.**

I. Introduction

THE window protocol is used to "control" the message exchange between two processes over two imperfect, unidirectional channels; it is characterized by the use of message sequence numbers to achieve reliable data transfer and the use of a "window" to control the flow of data. The window protocol has its roots in the alternating-bit protocol that was designed by Lynch [9], and Bartlett, Scantlebury, and Wilkinson [2], and was later studied by a host of researchers including Hailpern and Owicki [6], Gouda [5], and Chandy and Misra [4]. The generalization from the alternating-bit protocol to the window protocol was first suggested by Cerf and Kahn [3], and Stenning [14], and was later investigated by many researchers, most notably Knuth [7], and Shankar and Lam [11], [12]. Today, every major computer network (e.g., the ARPA network, the SNA network, and the ISO network standard) employs one or more versions of the window protocol.

The purpose of this paper is to develop a version of the window protocol that can tolerate message disorder (i.e., messages may be delivered out of the order in which they are sent) in addition to message loss, and where message sequence numbers are bounded (i.e., taken from a finite domain). Most existing protocols achieve one of these two goals but not both. For example, the window protocol of Stenning [14] can tolerate both message disorder and loss but assumes that message sequence numbers are unbounded, i.e., taken from the infinite domain of natural numbers. On the other hand, the window protocol of Cerf and Kahn [3] assumes that message sequence numbers are bounded and it can tolerate message loss, but it cannot tolerate message disorder.

The difficulty of achieving these two goals simultaneously is better explained by example. Consider the following scenario of a window protocol between two processes, called sender and receiver. (From this point and until the end of the introduction, the reader is assumed to be familiar with the go-back-N version of the window protocol as defined for instance in Stallings [13]. Appreciating the rest of the paper, however, does not require any prior knowledge of the window protocol.) The sender sends six data messages with sequence numbers ranging from 0 to 5; the receiver receives the messages 0 to 4 and acknowledges their reception by one acknowledgment message that has the sequence number 4. Later, the receiver receives the last data message whose sequence number is 5 and sends another acknowledgment message with the sequence number 5. Because of a message disorder, the sender receives only the second acknowledgment message, while the first acknowledgment remains in the channel from the receiver to the sender. Because the received acknowledgment has the sequence number 5, the sender recognizes that all the data messages that it has sent have been received by the receiver, and so proceeds to send new data messages. The unreceived acknowledgment remains in the channel from the receiver to the sender and can still be received at a later instant. Now, if the sequence numbers of data messages are bounded, the sender eventually sends new data messages with sequence numbers 0 to 4. It is possible that these messages are lost but the sender still receives the old acknowledgment from its input channel and recognizes wrongly that all these messages have been received correctly by the receiver.

There exist two versions of the window protocol where the two goals of bounded sequence numbers and tolerating message disorder are achieved [11]. [12]. [14]. In each case, however, the proposed protocol does not display all the nondeterminacy that is available in a regular window protocol. For instance, the selective-repeat protocol in [14] requires that every data message be acknowledged by a distinct acknowledgment message. It is clear in this case that the above scenario cannot occur. On the other hand, this is a severe restriction over the behavior of a regular window protocol, and can greatly reduce the protocol's performance.

The second protocol is due to Stenning [14] and later Lam and Shankar [11]. [12]. In it, the sending of a new data message depends not only on the availability of an "open window," as in the case of a regular window protocol, but also on some additional realtime constraint. In particular, a specified time period should elapse between the sending of two data messages with the same sequence number. The specified period guarantees that no copy of the first data message or its acknowledgment is still in transit between the sender and receiver before the sender sends the next data message with the same sequence number. This additional constraint may adversely affect the rate of data transfer in the event that a small domain of sequence numbers is used in the implementation of this protocol.

The window protocol described in this paper achieves the two goals of bounded sequence numbers and tolerating message disorder, without having a potential degradation of throughput.

Paper approved by the Editor for Communication Protocols of the IEEE Communications Society. Manuscript received March 29, 1989; revised January 24, 1990. This work was supported in part by ONR under Contract N00014-86-K-0763.

G. M. Brown is with the School of Electrical Engineering, Phillips Hall, Cornell University, Ithaca, NY 14853.

M. G. Gouda is with the Department of Computer Sciences, University of Texas at Austin, Austin TX 78712.

R. E. Miller is with the Department of Computer Science, University of Maryland, College Park, MD.

IEEE Log Number 9143095.

The basic idea of the protocol is quite straightforward. Each acknowledgment message has an associated pair of sequence numbers (m, n) where $m \leq n$, and acknowledges the reception of all data messages with sequence numbers ranging from m to n inclusive. Thus, the reception of a data message with sequence number k can be acknowledged by an acknowledgment message that has the pair (k, k). In general, however, one acknowledgment message can acknowledge the reception of any number of outstanding data messages. We refer to this method of acknowledgment as *block acknowledgment.*

If block acknowledgment is used in a window protocol, the above scenario will not lead to message loss as before. This is because the first acknowledgment message will have the pair $(0, 4)$ while the second will have the pair $(5, 5)$. Even if a message disorder causes the sender to receive the second acknowledgment before the first, the sender still has to receive the first acknowledgment before proceeding to send new data messages.

In our protocol, each data message that is received correctly is acknowledged exactly once by the receiver. If a data message or its acknowledgment is lost during transmission, the sender timesout and resends the data message. The correctness of the protocol requires that at most one copy of each data message or its acknowledgment is in transit at every instant. Thus, the timeout period should be chosen large enough to guarantee that a data message is resent (by the sender) only when the last copy of this message or its acknowledgment is lost during transmission.

Our protocol, like those of Stenning [14] and Lam and Shankar [11], [12], depends for its correctness on realtime constraints. But unlike those earlier protocols, our protocol resorts to the realtime constraints only when some message is lost—usually a rare occurrence. As long as sent messages are not lost, our protocol behaves exactly like a regular go-back-N window protocol except for sending two sequence numbers, instead of one, in every acknowledgment message.

In this paper, we define the protocol in some detail and give a proof of its correctness. This is accomplished in three steps. First, an initial version of the protocol that uses a simplified timeout action and unbounded sequence numbers is presented in Section II. A detailed proof for the correctness of this version is given in Section III. Second, the simplified timeout action in the protocol is replaced by a sophisticated one without disturbing the protocol's correctness in Section IV. Finally, in Section V, the unbounded sequence numbers are replaced by bounded ones while preserving the protocol's correctness. We conclude with some remarks concerning other variations of the protocol in Section VI.

II. A Window Protocol with Block Acknowledgments

The window protocol is used to control the message exchange between two processes over two unidirectional channels that may lose or reorder messages. One of the processes, named *sender* or S for short, sends data messages to the other process, named *receiver* or R for short, which then sends back acknowledgments. Each sent data message is stored in the channel from S to R until it is lost or received by R, and each sent acknowledgment is stored in the channel from R to S until it is lost or received by S. Each of the two channels is formally defined as a *set of messages* whose membership changes as new messages are sent into it or as old messages are lost or received from it.

In the protocol presented in this section, each new data message is assigned a new sequence number taken from the natural numbers. (Later, we modify this protocol to utilize sequence numbers from a finite domain.) Because each data message is uniquely associated with a sequence number, we define a data message to consist solely of its sequence number. This is a useful abstraction of the protocol since our concern is with the control issues involved in the data transfer and not with the data actually being transferred.

As a further abstraction, we assume that S has an infinite boolean array *ackd*[0..] in which it records those data messages that have been sent and later acknowledged by R and that R has an infinite boolean array *rcvd*[0..] in which it records those data messages that have been received. Assuming that these arrays are infinite greatly simplifies the reasoning about the protocol. It should be understood, however, that an implementation of the sender will require only a finite buffer space for storing data messages that have been sent, but not yet acknowledged, and an implementation of the receiver will require only a finite buffer space for storing those data messages that have arrived out of sequence.

The code for each process consists of a set of *actions* with the following syntax:

begin action [] ··· [] action **end**

Each action may be enabled or disabled depending upon the state of the system. An enabled action may be executed at any time with the restriction that the actions of the system (both R and S) are executed one at a time, i.e., atomically.

Each action has the form *guard* → *command*. The guard may be either a Boolean expression or a receive statement of the form *rcv x*. The commands are constructed from assignment statements and send statements of the form *send x* using sequencing, alternative, and iterative constructs.

An action in the sender's program (receiver's program) is *enabled* if its guard is a Boolean expression that evaluates to true or its guard is a receive statement and there is a message in the channel from R to S (or S to R).

Execution of an enabled action of $S(R)$ whose guard is a receive statement consists of receiving a message chosen at random from those in the channel from R to S (S to R) and then executing the action's command. Executing a send command in the sender's program (receiver's program) causes a message to be added to the channel from S to R (R to S).

The sender maintains a "window" of messages in transit. This window may vary in size, but at any instant has a maximum size of w. The boundaries of the sender's window are defined by na, the next message to be acknowledged, and ns, the next message to be sent. Each message with sequence number m where $na \leq m < ns$, is still outstanding, i.e., sent, but not yet acknowledged. Thus, whenever $na = ns$, there are no outstanding messages, and whenever $ns = na + w$ the window has its maximum size and no further messages may be sent until some outstanding messages are acknowledged. Whenever the window is smaller than its maximum size, the sender may send a new data message by executing the following action.

$$0: ns < na + w \rightarrow \textbf{send } ns;$$
$$ns := ns + 1$$

Each acknowledgment message consists of two numbers (m, n) to acknowledge all data messages with sequence numbers ranging from m to n inclusive. Whenever the sender receives an acknowledgment message, it records in the array *ackd* the acknowledged messages and modifies na to indicate the highest numbered

message that has not yet been acknowledged.

$$\begin{aligned} 1:\ \mathbf{rcv}(i,j) \rightarrow \mathbf{do}\ i \le j \rightarrow\ & ackd[i] := \text{true}; \\ & i := i+1 \\ [\!]\ ackd[na] \rightarrow\ & na := na+1 \\ \mathbf{od}\ & \end{aligned}$$

Because the channels are imperfect, messages may be lost and hence data messages may have to be retransmitted. The following action retransmits the outstanding data message with the lowest sequence number (*na*) whenever the Boolean expression *timeout* is true.

$$2: \text{timeout} \rightarrow\ \mathbf{send}\ na$$

We postpone defining the Boolean expression *timeout* until after presenting the receiver's code. The complete program for the sender can now be written as follows.

$$\begin{aligned} &\mathbf{process}\ S; \\ &\quad \mathbf{const}\ w : \mathbf{integer\ val}\quad /^*w > 0^*/ \\ &\quad \mathbf{var}\ ackd\ :\ \mathbf{array\ [integer]\ of\ Boolean\ init}\ false; \\ &\qquad na, ns, i, j : \mathbf{integer\ init}\ 0; \\ 0:\ &\mathbf{begin}\ ns < na + w \rightarrow \mathbf{send}\ ns; \\ &\qquad\qquad\qquad\qquad ns := ns+1 \\ 1:\ &[\!]\ \mathbf{rcv}(i,j) \qquad\qquad \rightarrow \mathbf{do}\ i \le j \rightarrow ackd[i] := \text{true}; \\ &\qquad\qquad\qquad\qquad\qquad\qquad\quad i := i+1 \\ &\qquad\qquad\qquad\qquad\quad [\!]\ ackd[na] \rightarrow na := na+1 \\ &\qquad\qquad\qquad\qquad\quad \mathbf{od} \\ 2:\ &[\!]\ \text{timeout} \qquad\qquad \rightarrow \mathbf{send}\ na \\ &\quad \mathbf{end} \end{aligned}$$

The receiver receives the data messages out of order, but only acknowledges them in order. The receiver maintains the sequence number of the next message to be accepted in *nr*, i.e., received and acknowledged. All data messages with sequence numbers less than *nr* have been accepted. While other data messages may have been received, no others have been acknowledged. The receiver accepts data messages in action 3. The reception of data message *v* is recorded in *rcvd*. If *v* has been accepted previously, then a duplicate acknowledgment is sent, otherwise the acknowledgment is postponed until all data messages with lower sequence numbers have been accepted.

$$\begin{aligned} 3:\ \mathbf{rcv}\ v \rightarrow\ & \mathbf{if}\ v < nr \rightarrow \mathbf{send}\ (v,v) \\ & [\!]\ v \ge nr \rightarrow rcvd[v] := \text{true} \\ & \mathbf{fi} \end{aligned}$$

The receiver attempts to acknowledge as many data messages as possible with a single block acknowledgment message. It uses *vr* to determine the upper bound of received data messages, i.e., all messages $m, nr \le m < vr$, have been received but not yet acknowledged. The receiver increments *vr* if the corresponding data message has been received.

$$4: rcvd[vr] \rightarrow vr := vr+1$$

Once the receiver has established that a block of data messages can be acknowledged, it does so by sending the acknowledgment $(nr, vr-1)$. The receiver then increases *nr* to reflect the upper bound of accepted messages

$$5: nr < vr \rightarrow \mathbf{send}\ (nr, vr-1); nr := vr$$

The complete program for the receiver can now be written as follows:

$$\begin{aligned} &\mathbf{process}\ R; \\ &\quad \mathbf{var}\ rcvd : \mathbf{array\ [integer]\ of\ Boolean\ init}\ false; \\ &\qquad nr, vr, v : \mathbf{integer\ init}\ 0; \\ 3:\ &\mathbf{begin}\ \mathbf{rcv}\ v \rightarrow \mathbf{if}\ v < nr \rightarrow \mathbf{send}\ (v,v) \\ &\qquad\qquad\qquad\qquad [\!]\ v \ge nr \rightarrow rcvd[v] := true \\ &\qquad\qquad\qquad\qquad \mathbf{fi} \\ 4:\ &[\!]\ rcvd[vr] \quad \rightarrow vr := vr+1 \\ 5:\ &[\!]\ nr < vr \quad \rightarrow \mathbf{send}(nr, vr-1); nr := vr \\ &\mathbf{end} \end{aligned}$$

In the introduction we showed that, for protocols in which messages may arrive out of order, data messages should not be retransmitted if a copy of the message or a copy of its acknowledgment still exists in transit. This can be accomplished by defining the Boolean expression *timeout* in *S* as follows:

$$\text{timeout} \equiv (na \ne ns) \wedge (C_{SR} = C_{RS} = \{\}) \wedge \neg rcvd[nr]$$

where C_{SR} denotes the set of messages in transit from *S* to *R* and C_{RS} denotes the set of messages in transit from *R* to *S*. Informally, the three conjuncts of this expression can be explained as: there are some outstanding data messages which have been sent by *S* and their acknowledgments have not been received by *S*, there are no data or acknowledgment messages in transit, and all received data messages have been acknowledged by *R*. Although we do not specifically address the implementation of *timeout*, a reasonable implementation would require a local timer for the sender and a mechanism for aging messages in transit, i.e., ensuring that they are eventually discarded if not received.

The correctness of this protocol is established in the next section. This proof of correctness will be needed in the following section to make the protocol have finite sequence numbers.

III. Proof of Correctness

We prove the safety, progress, and fault-tolerance properties of the protocol separately. First, we show that data messages are accepted by the receiver in the order sent (i.e., safety). We then show that the sender will be enabled continually to send the next data message in its input sequence and that each sent data message will eventually be received by the receiver (i.e., progress). Finally, we show that both safety and progress are still achieved even if sent messages can be lost or reordered (i.e., fault-tolerance).

We verify the safety properties of the protocol by presenting an appropriate invariant. An *invariant* is a state predicate that is true in the initial state and remains true after executing every action of the protocol. The invariant we present implies the following three properties: only data messages that have been sent are received,

$$(\forall\ m : rcvd[m] : m < ns),$$

a data message is accepted only if all prior data messages have been received,

$$(\forall\ m : \neg rcvd[m] : nr \le m),$$

and only data messages that have been accepted are acknowledged,

$$(\forall\, m : ackd[m] : m < nr).$$

To prove infinite progress of the protocol, we show that the sender will send new data messages infinitely often and that the receiver will accept new data messages infinitely often. This requires showing that each of the two actions 0 and 5 is executed infinitely often.

To prove fault-tolerance, we show that our proofs of safety and progress are still valid if sent messages can be lost or reordered.

A. Safety

We verify the three safety properties mentioned in Section III by presenting a single invariant. The invariant consists of three assertions, 6–8, which we present individually but which are mutually dependent for their invariance.

The variables *na*, *ns*, *nr*, and *vr* are related by a single assertion. The window of outstanding messages is bounded from below by *na* and from above by *ns*. Furthermore, at all times the maximum size of the window is *w*.

$$na \le ns \wedge$$
$$ns \le na + w$$

The receiver expects to receive data message *nr* which lies within the window. The receiver builds up acknowledgment messages with *vr*, hence all messages between *nr* and *vr* should have been received, but not yet acknowledged.

$$na \le nr \wedge$$
$$nr \le vr \wedge$$
$$vr \le ns$$

These requirements are summarized in assertion 6.

$$6\text{: } na \le nr \wedge$$
$$nr \le vr \wedge$$
$$vr \le ns \wedge$$
$$ns \le na + w$$

All messages less than *na* have been acknowledged (i.e., *ackd* is true), no message greater than or equal to *nr* has been acknowledged, and *na* has not been acknowledged. All received messages must have been sent (i.e., have values less than *ns*) and no message that has not been received should be acknowledged (i.e., all messages between *nr* and *vr* will be acknowledged). These requirements lead to the following assertion:

$$7\text{: } (\forall\, m : \neg ackd[m] : m \ge na) \wedge$$
$$(\forall\, m : ackd[m] : m < nr) \wedge$$
$$\neg ackd[na] \wedge$$
$$(\forall\, m : rcvd[m] : m < ns) \wedge$$
$$(\forall\, m : \neg rcvd[m] : m \ge vr)$$

Assertions 6 and 7 could be violated if an inappropriate message is received. We use the following notation to describe the messages in transit:

$^{\#}SR^m \equiv$ number of messages with sequence number m in C_{SR}

$^{\#}RS^m \equiv$ number of messages (x, y) in C_{RS} where $x \le m \le y$ holds.

To ensure the correctness of the protocol in Section II, we require only that no data message $m, m \ge ns$, and no acknowledgment message $(i, j), vr \le j$, be in transit; however, in order to modify the protocol to use a finite set of sequence numbers, we further require that there never be two messages in transit with the same sequence number. These requirements are captured by the following assertion:

$$8\text{: } (\forall\, m :: (^{\#}SR^m + {}^{\#}RS^m) \le 1) \wedge$$
$$(\forall\, m : {}^{\#}SR^m > 0 : m < ns \neg ackd[m]$$
$$\wedge\, (m < nr \vee \neg rcvd[m])) \wedge$$
$$(\forall\, m : {}^{\#}RS^m > 0 : m < nr \wedge \neg ackd[m]).$$

The system invariant consists of the conjunction of assertions 6–8. To prove that the resulting assertion is indeed an invariant, simply check that each of the actions 0–5 preserves it. For example, consider the first conjunct of assertion 6, $na \le nr$. Actions 0, 1, 3, and 4 alter neither *na* nor *nr* and hence preserve the validity of $na \le nr$. In action 2, message (i, j) is received by the sender. But since $i < nr$ and $j < nr$ (from 8), and *ack*[*na*] implies $na < nr$ (from 7), the loop of action 2 preserves $na \le nr$. Finally, in action 5, *nr* is increased and hence $na \le nr$ is preserved.

B. Progress

To prove indefinite progress of the protocol, we show that process *S* will send new data messages infinitely often and that process *R* will accept new data messages infinitely often. This is equivalent to showing that actions 0 and 5 are executed infinitely often, or that variable *ns* is incremented infinitely often. Because the four variables na, ns, nr, and vr are related by 6, namely,

$$na \le nr \le vr \le ns \le na + w$$

and because each of these variables can only be incremented by the processes, it is sufficient to show that the sum

$$na + ns + nr + vr$$

is incremented infinitely often.

At each reachable state of the system, at least one of the actions is enabled and can be executed. Moreover, each execution of action 0, 4, or 5 increments the sum. Therefore, it remains to show that executing actions 1, 2, and 3 will eventually lead to incrementing the sum.

Assume that actions 1, 2, and 3 start executing at some network state *s*. There are two cases to consider.

Case 0) Both Channels are Empty at s: In this case, only action 2 is enabled. Executing this action causes a data message with sequence number *na* to be sent into C_{SR} which enables action 3. Executing action 3 causes an acknowledgment message with the pair (*na*, *na*) to be sent into C_{RS}; this enables action 1 which increments *na* and hence the sum.

Case 1) At Least One of the Two Channels is not Empty at s: In this case, action 2 is not enabled, and only actions 1 and 3 can be executed. Because each execution of these two actions causes

one message to be received, the network will eventually reach a state in which both channels are empty (first C_{SR} is emptied and then C_{RS}). Starting from this state, Case 0) can be applied to show that eventually *na* will be incremented.

C. Fault-Tolerance

We have based our proofs of safety and progress on the assumption that C_{SR} and C_{RS} are both sets rather than sequences. Therefore, these proofs already take into account the possibility of "message disorder". The above invariant, namely, the conjunction of assertions 6, 7, and 8, is "insensitive" to message loss. Therefore, our proof of safety still holds if message loss is allowed. On the other hand, our proof of progress is not valid if sent messages can be lost frequently. However, by making the reasonable assumption, that there are long periods of time during which no sent message is lost, our proof of progress becomes valid and indeed establishes progress during those periods.

IV. A Window Protocol with Sophisticated Timeouts

The preceding protocol is designed under the optimistic, and in many cases realistic, assumption that transmission errors are rare. We have utilized this assumption in simplifying the timeout condition of the protocol. In particular this condition can be implemented using only one timer in process S. Process S need only keep track of the elapsed time period since it last sent a data message to R. When this time period exceeds a predefined value, indicating that no more messages are in transit between S and R, S timesout and resends message *na*.

Unfortunately, the simplicity of the timeout condition causes the protocol to be slow in its recovery from some types of errors. For example, if one acknowledgment message (m, n) is lost, process S has to timeout and resend each of the messages from m to n, one at a time, with each two successive messages separated by a *full* timeout period. Therefore, the above protocol needs to be modified slightly to speed up its recovery whenever transmission errors are not rare.

One possible modification is to replace the timeout action (2) in process S by

$$2' : \text{timeout}(i) \ \rightarrow \textbf{send}\ i$$

where *timeout*(i) is the following Boolean expression:

$$\begin{aligned} &(na \le i \wedge i < ns \wedge \neg ackd[i]) \wedge \\ &(\#SR^i = 0) \wedge \\ &(i < nr \vee \neg rcvd[i]) \wedge \\ &(\#RS^i = 0)s \end{aligned}$$

Recall that

$$\begin{aligned} \#SR^i &\equiv \text{number of messages with sequence number } i \text{ in } C_{SR} \\ \#RS^i &\equiv \text{number of messages } (x, y) \text{ in } C_{RS} \text{ where } x \le i \le y \text{ holds.} \end{aligned}$$

The first three conjuncts indicate that process S has sent a message i but not yet received an acknowledgment for it. The fourth conjunct indicates that message i is no longer in channel C_{SR}. The fifth conjunct indicates that process R cannot acknowledge message i (until process S resends it). The last conjunct indicates that no acknowledgment of message i is in channel C_{RS}.

Clearly, this timeout condition is more complicated than the original one. In particular, its implementation requires that process S has one independent timer for each outstanding message. However, the gain is in the speed of recovery: successive resendings of different messages do not have to be separated by any specific time period.

Next we establish the correctness of this protocol by an argument very similar to the one of our previous protocol.

A. Safety

Our invariant of the previous protocol, namely, the conjunction of assertions 6–8, is also an invariant of the current protocol. This can be established by showing that action 2′ does not falsify any of the assertions 6–8. The effect of action 2′ is to add one message to C_{SR}. Because assertions 6 and 7 do not mention C_{SR}, neither of them can be falsified by action 2′. Moreover, although assertion 8 does mention C_{SR}, the Boolean expression *timeout*(i) guarantees that adding message i to C_{SR} cannot falsify 8.

B. Progress

As part of action 2′, we have the following subaction in which *na* is substituting for i:

$$timeout(na) \ \rightarrow \ \textbf{send}\ na.$$

This subaction is similar to action 2 except for the Boolean expression *timeout*(na) replacing *timeout*. It is straightforward to show that the conjunction of the assertions 6, 7, 8, and timeout implies *timeout*(na). Thus, progress of the original protocol guarantees progress of the new protocol under action fairness, i.e., under the assumption that every enabled action eventually executes.

C. Fault Tolerance

The argument for the fault-tolerance of the current protocol is identical to the one of the previous protocol (Section III-C).

V. A Window Protocol with Finite Sequence Numbers

As a practical matter we would like to develop a protocol in which a finite set of sequence numbers is used by the sender and receiver. We show how to modify the window protocol of Section II so that it requires only a finite set of sequence numbers. The basic idea is quite simple. The sender and receiver continue to generate monotonically increasing sequence numbers internally, but rather than send the actual sequence number m, they send (m mod n) where n is a constant and (m mod n) is the remainder obtained when m is divided by n. Provided that the process receiving (m mod n) has a way to reconstruct m, no information is lost. We now show how to perform this reconstruction.

The only action of the sender's program that accesses the contents of a received message is 1. From assertions 6 and 8 we conclude that

$$9:\ 0 \le na \le i < na + w \qquad \text{(in 1)}$$

$$10:\ 0 \le na \le j < na + w \qquad \text{(in 1)}$$

Similarly, the only action of the receiver's program that accesses the contents of a received message is 3 and from assertions 6 and 8 we conclude that

$$11:\ 0 \le \max(0, nr - w) \le v < nr + w \qquad \text{(in 3)}$$

Assertions 9–11 give us the information required to reconstruct i, j, and v from (i mod n), (j mod n) and (v mod n), respectively, where n is a constant still to be selected. In particular, for any x and y such that

$$12:\ 0 \leq x \leq y < x + n$$

we have

$$13:\ ((x \text{ div } n) = (y \text{ div } n)) \equiv ((y \bmod n) \geq (x \bmod n))$$

and

$$14:\ ((1 + (x \text{ div } n)) = (y \text{ div } n)) \equiv ((y \bmod n) < (x \bmod n))$$

where (m div n) is the integer resulting from dividing m by n. (Notice the similarity between 12 and each of the assertions 9–11.)

We will use 13 and 14 to create a function f such that

$$f(x, y) \equiv y$$

and f accesses only x and (y mod n). From 13 and 14, we can derive (y div n) from x and (y mod n). Therefore, we define

$$\begin{aligned} f(x,y) \equiv\ & \textbf{if} (y \bmod n) \geq (x \bmod n) \\ & \rightarrow (y \bmod n) + n(x \text{ div } n) \\ & [\!] (y \bmod n) < (x \bmod n) \\ & \rightarrow (y \bmod n) + n(1 + (x \text{ div } n)) \\ & \textbf{fi}. \end{aligned}$$

If we then define

$$n \equiv 2w$$

then

$$f(na, i) = i \quad \text{(in 1)}$$
$$f(na, j) = j \quad \text{(in 1)}$$

and

$$f(\max(0, (nr - w)), v) = v \quad \text{(in 3)}.$$

Given the definition of f, we can now show that actions 1 and 3 can be modified so that they only "examine" the sequence numbers in the received messages modulo n. The preceding development demonstrates that such a modification can be performed without altering either the safety or progress properties of the protocol.

$$\begin{aligned} 1: \textbf{rcv}(i,j) \rightarrow \textbf{do}\ & i \leq j \rightarrow ackd[i] := \text{true}; \\ & \quad i := i + 1 \\ & [\!]\ ack[na] \rightarrow na := na + 1 \\ & \textbf{od} \end{aligned}$$

by

$$\begin{aligned} 1': \textbf{rcv}(i,j) \rightarrow \textbf{do}\ & f(na,i) \leq f(na,j) \rightarrow\ ackd[f(na,i)] := \text{true}; \\ & \quad i := (i+1) \bmod \text{n} \\ & [\!]\ ackd[na] \quad\ \rightarrow\ na := na + 1 \\ & \textbf{od} \end{aligned}$$

and

$$\begin{aligned} 3: \textbf{rcv}\ v \rightarrow\ & \textbf{if}\ v < nr \rightarrow \textbf{send}(v, v) \\ & [\!]\ v \geq nr \rightarrow \textbf{rcvd}[v] := \text{true} \\ & \textbf{fi} \end{aligned}$$

by

$$
\begin{aligned}
3': \textbf{rcv}\ v \rightarrow\ & \textbf{if} f(\max(0,(nr-w)),v) < nr \rightarrow \textbf{send}(v,v) \\
& [\!] f(\max(0,(nr-w)),v) \geq nr \ \rightarrow \textbf{rcvd}[f(\max(0,(nr-w)),v)] := \text{true} \\
& \textbf{fi}
\end{aligned}
$$

Since the sender and receiver now only examine messages modulo n (i.e., in f), they need only transmit messages with sequence numbers modulo n. This leads to the following programs (note $(x \bmod n) \equiv ((x \bmod n) \bmod n)$).

```
const w : integer val;   /*w > 0*/
  n : integer 2w
process S;
  var ackd : array [integer] of Boolean init false;
      na, ns, i, j : integer init 0;
  begin ns < na + w → send ns mod n;
                      ns := ns + 1
  [] rcv(i, j)        → do f(na, i) ≤ f(na, j) → ackd[f(na, i)] := true;
                                                 i := (i + 1) mod n
                        [] ackd[na]            → na := na + 1
                        od
  [] timeout(i)       → send i mod n
  end
process R;
  var rcvd : array[integer] of Boolean init false
      nr, vr, v : integer init 0;
  begin
  rcv v → if f(max(0, (nr − w)), v) < nr → send (v, v)
          [] f(max(0, (nr − w)), v) ≥ nr → rcvd[f(max(0, ((nr − w)), v)] := true
          fi
  [] rcvd[vr] → vr := vr + 1
  [] nr < vr → send((nr mod n), ((vr − 1) mod n)); nr := vr
  end
```

Although the preceding protocol utilizes a finite range of sequence numbers, it still requires unbounded storage for each process because it uses unbounded integers (*na*, *ns*, *nr*, and *vr*) and unbounded arrays (*ackd*, and *rcvd*). We can further modify this protocol so that each process uses bounded storage. First, the unbounded arrays *ackd* and *rcvd* are replaced by finite arrays. Then the comparisons, $ns < na+w, i < j, v < nr$, and $nr < vr$ are modified to access *ns*, *na*, *nr*, and *vr* modulo *n* (recall that $n = 2w$). Finally, the actions that increment *ns*, *na*, and *vr* are modified so that they increment modulo *n*. In the interest of brevity we only sketch the development of these modifications.

Consider the array *ackd* which is used to record those messages that have been both sent and acknowledged. From assertions 6 and 7 of the invariant we see that for all messages $i, i < na (\text{or } ns \leq i), ack[i](\neg ackd[i])$ holds. Thus, we only need *w* storage locations for the subarray $ackd[na \ldots ns-1]$. Similarly, we need only *w* storage locations for the subarray $rcvd[vr \ldots ns-1]$. To implement these modifications, each of the actions $1', 3'$, and 4 should be altered so that arrays *ackd* and *rcvd* are accessed modulo w, $ackd[na \bmod w]$ is set to false in action $1'$, and $rcvd[vr \bmod w]$ is set to false in action 4.

As an example of how the comparisons in the protocol are modified, consider the comparison $i \leq j$ in action 1. An invariant of the loop in action 1 is

$$i \leq j + 1$$

therefore

$$i \leq j \equiv i \neq j + 1.$$

From 9 and 10 we conclude that

$$i \leq j + 1 < i + n$$

is an invariant of the loop. Finally, from 13 we conclude that

$$
\begin{aligned}
i \ &\neq j+1 \equiv i \bmod n \neq (j+1) \bmod n \equiv i \bmod n \\
&\neq ((j \bmod n)+1) \bmod n.
\end{aligned}
$$

The other comparisons in the protocol can be modified similarly. Once these modifications have been performed, *ns*, *na*, *nr*, and *vr* are only accessed modulo *n* by the programs. Hence, all additions in the programs may be performed modulo *n*.

VI. Concluding Remarks

To the best of our knowledge the concept of block acknowledgment is new for window protocols. It provides for a more definite form of acknowledgment than that in earlier window protocols, with the small added expense of needing two sequence numbers in the acknowledgment rather than one. As we have shown this modification of the acknowledgment enables us to overcome message loss and message reordering in the transmission medium while maintaining the throughput advantages of traditional window protocols.

The window protocol with block acknowledgment combines the desirable features of two sell-known versions of the window protocol, namely, go-back-*N* with that of selective-repeat [13]. In particular, it can tolerate message disorder (selective-repeat) and a single message can acknowledge a large number of data messages (go-back-*N*). In fact, selective-repeat and go-back-*N* are special cases of block acknowledgment where only acknowledgments of the form (v, v) and $(0, v)$ are sent, respectively.

Our window protocol relies upon the implementation of accurate timeout mechanisms. This is a requirement of all practical protocols which use bounded sequence numbers and which can tolerate both message loss and disorder. While there exist window protocols which use finite sequence numbers and do not require accurate timeouts, these are largely of theoretical interest [1], [10]. Mansour and Schieber have shown that in such a protocol the number of messages required to send *n* data elements is unbounded by any function of *n* when fewer than *n* sequence numbers are used [10]. Lynch previously showed that without accurate timeouts and with the requirement that only a bounded number of messages be sent, no [window] protocol can solve this problem [8].

The window protocols we have described in Sections II and IV utilize the concept of block acknowledgment in a very straightforward way. Yet, since block acknowledgment provides an exact acknowledgment of those messages that have been received, this opens up the possibility of utilizing any positions that have been acknowledged for transmission of new messages, even though some earlier messages in different positions have not yet been acknowledged. For example, suppose message 0 through 5 were sent, but only messages 3 through 5 were acknowledged. It would then be possible, through a more complicated protocol design, to reuse positions 3 through 5 for sending more messages before messages 0, 1, and 2 were received. To realize such a protocol the sender and receiver would have to remember more information about which messages had been sent and not yet received, and the protocol designs for the sender and receiver would have to be more complex. Clearly, there is some tradeoff here between the added complexity versus the potential gain in performance by more aggressive reuse of acknowledgment message positions.

For simplicity, we have assumed that the window size is fixed throughout this paper. It is possible, however, to extend all our protocols to have variable size windows; see for instance [12].

Acknowledgment

We are grateful to C. Edmondson, L. Landwebber, and the referees for their comments to earlier drafts of this paper; their suggestions have led to an improvement in the final protocol.

References

[1] H. Attiya, M. J. Fischer, D. Wang, and L. D. Zuck. "Reliable communication using unreliable channels," unpublished manuscript.
[2] K. A. Bartlett, R. A. Scantlebury, and P. Wilkinson, "A note on reliable full duplex transmission over half-duplex links," *Commun. ACM,* vol. 12, pp. 260–261, 1969.
[3] V. G. Cerf and R. E. Kahn, "A protocol for packet network intercommunication," *IEEE Trans. Commun.,* vol. COM-22, pp. 637–648, 1974.
[4] K. M. Chandy and J. Misra, *Parallel Program Design: A Foundation.* New York: Addison-Wesley, 1988, Ch. 17.
[5] M. G. Gouda, "On a simple protocol whose proof is not: The state machine approach," *IEEE Trans. Commun.* vol. COM-33, pp. 380–382, 1985.
[6] B. Hailpern and S. S. Owicki, "Modular verification of computer communication protocols," *IEEE Trans. Commun.,* vol. COM-31, pp. 56–68, 1983.
[7] D. E. Knuth, "Verification of link level protocols," *B.I.T.* vol. 21, pp. 31–36, 1981.
[8] N. A. Lynch, Y. Mansour, and A. Fekete, "Data link layer: Two impossibility results," in *Proc. 7th ACM Symp. Principles Distrib. Comput.,* 1988, pp. 149–170.
[9] W. C. Lynch, "Reliable full-duplex file transmission over half-duplex telephone lines," *Commun. ACM,* vol. 11, pp. 407–410, 1968.
[10] Y. Mansour and B. Schieber, "The intractability of bounded protocols for non-FIFO channels," in *Proc. 8th ACM Symp. Principles Distrib. Comput.,* 1989, pp. 59–72.
[11] A. U. Shankar and S. S. Lam, "A stepwise refinement for protocol construction," Dep. Comput. Sci., Univ. Maryland, Tech. Rep. CS-TR-1812.1, Mar. 1987, revised Mar. 1989.
[12] A. U. Shankar, "Verified data-transfer protocols with variable flow control," *ACM Trans. Comput. Syst.,* vol. 7, no. 3, pp. 281–316, 1989.
[13] W. Stallings, *Data and Computer Communications.* New York: Macmillan, 1988, 2nd ed.
[14] N. V. Stenning, "A data transfer protocol," *Comput. Networks,* vol. 1, pp. 99–110, 1976.

Geoffrey M. Brown (S'82–M'87) received the B.S. degree in engineering from Swarthmore College in 1982, the M.S. degree in electrical engineering from Stanford University in 1983, and the Ph.D. degree in electrical engineering from the University of Texas at Austin in 1987.

He is currently an Assistant Professor of Electrical Engineering at Cornell University, Ithaca, NY. His research interests include distributed systems, communication protocols, and formal methods for hardware design. He was employed by Motorola Semiconductor during 1983–1984.

Dr. Brown was awarded a Presidential Young Investigator Award in 1990.

Mohamed G. Gouda has received five degrees in engineering (B.Sc. Cairo University 1968), mathematics (B.Sc. Cairo University 1971, M.A. York University 1972), computing science (M.Sc. and Ph.D. University of Waterloo 1973 and 1977).

He has worked for Honeywell Corporation (1977–1980), MCC (Summer 1986), and Bell Labs (Summer 1987). Since 1980, he has been with the Department of Computer Sciences, University of Texas at Austin, where he is currently an Associate Professor.

Dr. Gouda is the Editor-in Chief of the journal *Distributing Computing.*

Raymond E. Miller (S'54–M'58–F'70) received the B.S. degree in mechanical engineering from the University of Wisconsin, Madison, and the B.S. in electrical engineering, the M.S. in mathematics, and the Ph.D. in electrical engineering, all from the University of Illinois, Urbana.

He is a Professor of Computer Science at the University of Maryland, College Park, and Director of the USRA Center of Excellence in Space Data and Information Sciences at NASA Goddard Space Flight Center, Greenbelt, MD.

From 1980 to 1989 he was Director and Professor of the School of Information and Computer Science at the Georgia Institute of Technology. Prior to that he was employed by IBM for over 30 years, most of this time as a Research Staff Member at the IBM Research Center in Yorktown Heights, NY where he held a number of technical management positions. His research areas include theory of computation, machine organization, parallel computation, and communication protocols.

Dr. Miller has written over 70 papers, authored a two-volume book on switching theory, served as editor for a book on computer complexity, and edited a book series on foundations of computer science. He is a Fellow of AAAS, a member of ACM and has been active in numerous ACM capacities including being Editor-in-Chief of the *JOURNAL OF THE ACM* from 1972–1976, a member of the ACM Council for six years, an ACM National Lecturer for 1982–1983, and a member of the AFIPS Board of Directors for four years. Currently he is a member of the Computing Research Board, the IEEE Computer Society Board of Governors, and an ACM Representative on the Computing Sciences Accreditation Board. He was a member of the NSF Advisory Committee for Computer Research from 1982 to 1985, serving as Chairman for 1983–1984. He has taught at numerous institutions including Caltech, New York University, Yale, University of California at Berkeley, the Polytechnic Institute of New York, Georgia Tech, and the University of Maryland.