BLOCK ACKNOWLEDGEMENT: REDESIGNING THE WINDOW PROTOCOL

G. M. Brown,[1] M. G. Gouda,[2] and R. E. Miller[3]

Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712-1188

TR-89-02 March 1989

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 acknowledgement, called *block acknowledgement*, where each acknowledgement 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 acknowledgement, the proposed protocol achieves the two goals while maintaining the same data transmission capability of the traditional window protocol.

[1] School of Electrical Engineering, Cornell University.
[2] Work supported in part by Office of Naval Research Contract N00014-86-K-0763.
[3] School of Information and Computer Science, Georgia Institute of Technology.

1 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 [7], and Bartlett, Scantlebury, and Wilkinson [1], and was later studied by a host of researchers including Hailpern and Owicki [5], Gouda [4], and Chandy and Misra [3]. The generalization from the alternating-bit protocol to the window protocol was first suggested by Cerf and Kahn [2], and Stenning [10], and was later investigated by many researchers, most notably Knuth [6], and Shankar and Lam [8]. 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 [10] 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 [2] 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 [9]. 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 acknowledgement message that has the sequence number 4. Later, the receiver receives the last data message whose sequence number is 5 and sends another acknowledgement message with the sequence number 5. Because of a message disorder, the sender receives only the second acknowledgement message, while the first acknowledgement remains in the channel from the receiver to the sender. Because the received acknowledgement 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 acknowledgement 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 acknowledgement 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 [8,10]. 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 [10] requires that every data message be acknowledged by a distinct acknowledgement 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 by Lam and Shankar [8]. 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 acknowledgement 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. The basic idea of the protocol is quite straightforward. Each acknowledgement 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 acknowledgement 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 acknowledgement as *block acknowledgement.*

If block acknowledgement is used in a window protocol, the above scenario will not lead to message loss as before. This is because the first acknowledgement 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 acknowledgement before the first, the sender still has to receive the first acknowledgement before proceeding to send new data messages. Note that a message is never acknowledged more than once.

In this paper, we define the protocol in some detail and give a proof of its correctness. This is accomplished in two steps. First, a version of the protocol assuming unbounded sequence numbers is presented in Section 2, and a proof of its correctness is given in Section 3. Our proof shows that bounding the sequence numbers by some specific bound will not affect the protocol's correctness. This observation is utilized in Section 4 to develop a version of the protocol that uses bounded sequence numbers. We conclude with some remarks concerning different variations of the protocol in Section 5.

2 A Window Protocol with Block Acknowledgements

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 acknowledgements. Each sent data message is stored in the channel from S to R until it is lost or received by R, and each sent acknowledgement 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* $\rightarrow$ *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: $ns < na + w$  $\rightarrow$  send $ns$;
                        $ns := ns + 1$
```

Each acknowledgement 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 acknowledgement 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.

```
1:  rcv(i,j)    →  do $i \le j$     →  ackd[i] := true;
                                      $i := i + 1$
                   [] ackd[na] →  $na := na + 1$
                   od
```

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:  timeout  →  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.

```
process S;
    const w: integer  val            /* $w > 0$ */
    var    ackd : array [integer] of boolean init false;
           na, ns, i, j : integer init 0;

0:  begin $ns < na + w$   →   send ns;
                              $ns := ns + 1$
1:  []  rcv(i,j)          →   do $i \le j$    →  ackd[i] := true;
                                                 $i := i + 1$
                              [] ackd[na] →  $na := na + 1$
                              od
2:  []  timeout           →   send na
    end
```

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 acknowledgement is sent, otherwise the acknowledgement is postponed until all data messages with lower sequence numbers have been accepted.

```
3:  rcv $v$                 $\rightarrow$  if $v < nr \rightarrow$ send $(v, v)$
                                [] $v \geq nr \rightarrow rcvd[v] :=$ true
                                fi
```

The receiver attempts to acknowledge as many data messages as possible with a single block acknowledgement message. It uses *vr* to determine the upper bound of received data messages, i.e all messages m, $nr \leq m < vr$, 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 acknowledgement $(nr, vr -1)$. The receiver then increases *nr* to reflect the upper bound of accepted messages

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

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

```
process $R$;
    var    $rcvd$ : array[integer] of boolean init false;
           $nr, vr, v$ : integer init 0;

3:  begin rcv $v$           $\rightarrow$  if $v < nr \rightarrow$ send $(v, v)$
                                [] $v \geq nr \rightarrow rcvd[v] :=$ true
                                fi
4:  [] $rcvd[vr]$           $\rightarrow$  $vr := vr + 1$
5:  [] $nr < vr$            $\rightarrow$  send$(nr, vr$-1$)$; $nr := vr$
    end
```

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 acknowledgement still exists in transit. This can be accomplished by defining the boolean expression *timeout* in *S* as follows:

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

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 acknowledgements have not been received by S, there are no data or acknowledgement 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.

3 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.

3.1 Safety

We verify the three safety properties mentioned in Section 3 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*.

$$\begin{aligned}&na \leq ns \ \wedge\\&ns \leq na + w\end{aligned}$$

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

$$\begin{aligned}&na \leq nr \ \wedge\\&nr \leq vr \ \wedge\\&vr \leq ns\end{aligned}$$

These requirements are summarized in assertion 6.

$$\begin{aligned}6:\ &na \leq nr \ \wedge\\&nr \leq vr \ \wedge\\&vr \leq ns \ \wedge\\&ns \leq na + w\end{aligned}$$

All messages less than *na* have been acknowledged (i.e. *ackd* is true), and no message greater than or equal to *nr* has 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: $(\forall\, m : \neg ackd[m] : m \geq na) \wedge$
$(\forall\, m : ackd[m] : m < nr) \wedge$
$(\forall\, m : rcvd[m] : m < ns) \wedge$
$(\forall\, m : \neg rcvd[m] : m \geq vr)$

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 $x \leq m \leq y$ holds.

To ensure the correctness of the protocol in Section 2, we require only that no data message m, $m \geq ns$, and no acknowledgement message (i,j), $vr \leq j$, 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: $(\forall\, m :: (\#_{SR}m + \#_{RS}m) \leq 1) \wedge$
$(\forall\, m : \#_{SR}m > 0 : m < ns \wedge \neg ackd[m] \wedge (m < nr \vee \neg rcvd[m])) \wedge$
$(\forall\, m : \#_{RS}m > 0 : m < nr \wedge \neg ackd[m])$

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, $na \leq nr$. Actions 0, 1, 3, and 4 alter neither na nor nr and hence preserve the validity of $na \leq nr$. 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 $na \leq nr$. Finally, in action 5, nr is increased and hence $na \leq nr$ is preserved.

3.2 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

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

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 acknowledgement 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.

3.3 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.

4 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 2 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

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

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

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

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

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

$$12: \quad 0 \le x \le y < x + n$$

we have

$$13: \quad ((x \text{ div } n) = (y \text{ div } n)) \equiv ((y \text{ mod } n) \ge (x \text{ mod } n))$$

and

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

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

```
f(x,y) ≡   if $(y\ mod\ n) \geq (x\ mod\ n) \rightarrow (y\ mod\ n) + n(x\ div\ n)$
           [] $(y\ mod\ n) < (x\ mod\ n) \rightarrow (y\ mod\ n) + n(1 + (x\ div\ n))$
           fi
```

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

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

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.

```
1:  rcv(i,j) → do $i \leq j$ → $ackd[i]$ := true;
                        $i := i + 1$
               [] $ack[na] \rightarrow na := na + 1$
               od
```

by

```
1': rcv(i,j) → do $f(na,i) \leq f(na,j)$ → $ackd[f(na,i)]$ := true;
                                   $i := (i + 1)\ mod\ n$
               [] $ackd[na]$        → $na := na + 1$
               od
```

and

```
3:  rcv v  →  if v < nr →  send (v, v)
              [] v ≥ nr  →  rcvd[v] := true
              fi
```

by

```
3': 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
```

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             →   send na mod n
    end
```

```
process $R$;
    var    $rcvd$ : array[integer] of boolean init false;
           $nr, vr, v$ : integer init 0;
    begin
    rcv $v$     $\rightarrow$ if $f(\max(0,(nr-w)),v) < nr \rightarrow$ send $(v, v)$
                 [] $f(\max(0,(nr-w)),v) \geq nr \rightarrow rcvd[f(\max(0,((nr-w)),v)] :=$ true
                 fi
    [] $rcvd[vr] \rightarrow vr := vr + 1$
    [] $nr < vr \rightarrow$ send$((nr \bmod n), ((vr-1) \bmod n))$; $nr := vr$
    end
```

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

Although the preceeding 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 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, $i < na$ (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

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

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

$$i \neq j+1 \equiv \quad i \bmod n \neq (j{+}1) \bmod n \quad \equiv \quad i \bmod n \neq ((j \bmod n) + 1) \bmod n$$

The other comparisions 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*.

5 Concluding Remarks

To the best of our knowledge the concept of block acknowledgement is new for window protocols. It provides for a more definite form of acknowledgement than that in earlier window protocols, with the small added expense of needing two sequence numbers in the acknowledgement 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 acknowledgement combines the desirable features of two well known versions of the window protocol, namely go-back-N with that of selective-repeat [10]. 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 acknowledgement where only acknowledgments of the form (v,v) and (0,v) are sent, respectively.

The window protocols we have described in Sections 2 and 4 utilize the concept of block acknowledgement in a very straightforward way. Yet, since block acknowledgement provides an exact acknowledgement 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 acknowledgement message positions.

Acknowledgement: We are grateful to Chris Edmondson for her comments to an earlier draft of this paper; her suggestions have led to an improvement in the final protocol.

References

[1] K. A. Bartlett, R. A. Scantlebury, and P. Wilkinson, "A Note on Reliable Full Duplex Transmission over Half-Duplex Links," *Communications of the ACM*, vol. 12, pp. 260-261, 1969.

[2] V. G. Cerf and R. E. Kahn, "A Protocol for Packet Network Intercommunication," *IEEE Transactions on Communications*, vol. com-22, no. 5, pp. 637-648, 1974.

[3] K. M. Chandy and J. Misra, *Parallel Program Design: A Foundation*, Chapter 17, Addison–Wesley Publishing Co., 1988.

[4] M. G. Gouda, "On a Simple Protocol Whose Proof in Not: The State Machine Approach," IEEE Transactions on Communications, vol. com-33, no. 4, pp. 380-382, 1985.

[5] B. Hailpern and S. S. Owicki, "Modular Verification of Computer Communication Protocols," *IEEE Transactions on Communications*, vol. com-31, no. 1, pp. 56-68, 1983.

[6] D. E. Knuth, "Verification of Link Level Protocols," *BIT*, vol. 21, pp. 31-36, 1981.

[7] W. C. Lynch. "Reliable Full-Duplex File Transmission over Half-Duplex Telephone Lines," *Communications of the ACM*, vol. 11, pp. 407-410, 1968.

[8] A. U. Shankar and S. S. Lam, "Time-Dependent Distributed Systems: Proving Safety, Liveness, and Real-Time Properties," *Distributed Computing*, vol. 2, no. 2, pp. 61-79, 1987.

[9] W. Stallings, *Data and Computer Communications, 2nd edition*, New York, Macmillan, 1988.

[10] N. V. Stenning, "A Data Transfer Protocol," *Computer Networks*, vol. 1, pp. 99-110, 1976.