Redesigning the Window Protocol: the Block Acknowledgment Revisited

Anneke A. Schoone

RUU-CS-92-04
February 1992

Utrecht University

Department of Computer Science

Padualaan 14, P.O. Box 80.089,
3508 TB Utrecht, The Netherlands,
Tel. : ... + 31 - 30 - 531454

Redesigning the Window Protocol: the Block Acknowledgment Revisited

Anneke A. Schoone

Technical Report RUU-CS-92-04
February 1992

Department of Computer Science
Utrecht University
P.O.Box 80.089
3508 TB Utrecht
The Netherlands

ISSN: 0924-3275

Redesigning the Window Protocol: the Block Acknowledgment Revisited

Anneke A. Schoone*

Department of Computer Science, Utrecht University,
P.O. Box 80.089, 3508 TB Utrecht, The Netherlands

Email: anneke@cs.ruu.nl

Abstract

Brown, Gouda, and Miller [1] claim that they have redesigned the window protocol such that it tolerates both message loss and message disorder, while using only bounded sequence numbers. They suggest to implement the timeout condition of the protocol by means of timers. We analyze the consequences of an implementation with timers and show that it is not possible to implement the protocol as specified by means of timers. This is due to the fact that in the original protocol it is not observable for the sender whether the timeout condition holds or not. We slightly modify the protocol such that an implementation with timers is possible, and prove that it is correct.

1 Introduction

Brown, Gouda, and Miller [1] redesigned the window protocol to be able to tolerate both message loss and message disorder, where sequence numbers are taken from a finite domain. As the protocol tolerates message loss, it contains a possibility for retransmission of messages upon "timeout". In their article, the authors present three versions of the protocol. The first has a simple timeout condition and unbounded sequence numbers, the second has more sophisticated timeout conditions and unbounded sequence numbers, while the third version contains bounded sequence numbers.

The conditions derived for "timeout" in both cases ensure a correct operation of the protocol, however, whether the Boolean expression defined as "timeout" evaluates to true or not, is not directly observable for the sender process. The authors propose implementations of the timeout conditions by means of timer(s) in the

*The work of A.A. Schoone is supported by the ESPRIT II Basic Research Actions Program of the EC under contract no. 3075 (project ALCOM).

sender process only. We show that these implementations cannot meet the claimed performance of the protocol and suggest implementations with additional timer(s) in the receiver process. For the second version, we show that it is not possible to give any implementation with timers of the protocol as specified. We also show that any attempt at such an implementation can lead to an erroneous execution. Thus we conclude that the protocol is *not* correct, in spite of the fact that the authors give a proof of correctness in [1]. Hence we have changed the specification of the protocol slightly by relaxing the timeout condition to arrive at a protocol that can be implemented by means of timers. We then prove that this implementation is correct by means of an invariant. We will use unbounded sequence numbers in this note for clarity's sake, as the transition to bounded sequence numbers is independent of the implementation of "timeout" and exactly the same as in the original article.

The paper is structured as follows. Although we assume that the reader is familiar with the original article, we give a short overview of the original protocol together with the invariant that implies its safety in Section 2. Section 3 discusses the first version of the protocol with a simple timeout condition and the change we propose. Section 4 discusses the version of the protocol with sophisticated timeout conditions, an erroneous execution which can occur if one tries to implement the protocol with timers, and the proposed changes which lead to a slightly different protocol. Finally, Section 5 contains a proof that the proposed implementation with timers of the repaired protocol is correct.

2 The Block Acknowledgment Protocol

The window protocol described by Brown, Gouda, and Miller in [1] is based on a new method of acknowledgment, called *block acknowledgment*: each acknowledgment message has two numbers n and m to acknowledge all data messages with sequence numbers ranging from n to m. The protocol is developed in three versions. The first uses a simple timeout condition, while the second has more sophisticated timeout conditions. These versions use unbounded sequence numbers and arrays to facilitate reasoning about the protocol. The third, final, version uses bounded sequence numbers and arrays.

The window protocol is used to control the message exchange between two processes over two unidirectional channels that may lose or reorder messages. The process named S sends data messages to the process named R, which then sends back acknowledgments. The channel from S to R, C_{SR}, is modeled as a multiset of messages, to which messages are added when sent by S, and from which messages are deleted when lost or received by R. Likewise, the channel from R to S is modeled by the multiset C_{RS}. Each new data message is assigned a new sequence number from the natural numbers, and as we are not interested in the actual data transferred, a data message is identified with its sequence number. We assume that S has an infinite boolean array $ackd[0\ldots]$ in which it records which data messages

have been acknowledged by R. We assume that R has an infinite boolean array $rcvd[0\ldots]$ in which it records which data messages it has received. The sender S maintains a window of messages in transit, which has a maximum size of w. It is bounded by na, the next message to be acknowledged, and ns, the next message to send. An acknowledgment message consists of two numbers (n, m) to acknowledge all data messages between n and m, inclusive. The receiver R maintains the number of the next message to rceive and acknowledge in nr. The variable vr is used to determine how many data messages can be acknowledged in one acknowledgment message.

The code of a process consists of a set of *actions* with the syntax: **begin** action [] action [] action **end** . An action has the following form: *guard* $\rightarrow$ *command.* If the guard evaluates to true, the action is enabled, otherwise it is disabled. Enabled actions are executed atomically and in a fair way. A guard of the form **rcv** evaluates to true if there is a message to receive in the channel. The protocol is 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** $ns < na + w \rightarrow$ **send** ns;
    $ns := ns + 1$
1: [] **rcv**(i, j) $\rightarrow$ **do** $i \leq j \rightarrow$ $ackd[i] := true$;
    $i := i + 1$
    [] $ackd[na] \rightarrow na := na + 1$
    od
2: [] `timeout` $\rightarrow$ **send** na
  end

For the case of sophisticated timeout conditions, action 2 is replaced by action 2′:
2′: [] `timeout`(i) $\rightarrow$ **send** i

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

The boolean expressions `timeout` and `timeout`(i) are defined as follows:

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

$$\texttt{timeout}(i) \stackrel{\text{def}}{=} (na \leq i \wedge i < ns \wedge \neg ackd[i]) \wedge (\#SR^i = 0) \wedge (i < nr \vee \neg rcvd[i]) \wedge (\#RS^i = 0)$$

where

$\#SR^i \stackrel{\text{def}}{=}$ number of messages with sequence number i in C_{SR}

$\#RS^i \stackrel{\text{def}}{=}$ number of messages (n, m) with $n \leq i \leq m$ in C_{RS}.

For the proof of correctness of the protocol, safety, progress, and fault–tolerance are proved separately. The safety of the protocol follows from the invariant which is the conjunct of assertions 6, 7, and 8.

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

7: $(\forall m : \neg ackd[m] : m \geq na) \wedge (\forall m : ackd[m] : m < nr) \wedge \neg ackd[na] \wedge$
$(\forall m : rcvd[m] : m < ns) \wedge (\forall m : \neg rcvd[m] : m \geq vr)$

8: $(\forall m :: (\#SR^m + \#RS^m) \leq 1) \wedge$
$(\forall m : \#SR^m > 0 : m < ns \wedge \neg ackd[m] \wedge (m < nr \vee \neg rcvd[m])) \wedge$
$(\forall m : \#RS^m > 0 : m < nr \wedge \neg ackd[m])$

Assertion 9 establishes that it is safe to use sequence numbers and arrays modulo n if $n \geq 2w$:

9: $(\forall m : \#SR^m > 0 : nr - w \leq m < nr + w) \wedge$
$(\forall m : \#RS^m > 0 : na \leq m < na + w)$

The authors suggest that the timeout conditions mentioned in the protocol be implemented by means of timers. They claim that the proposed protocol tolerates message loss and message disorder while maintaining the same data transmission capability of the traditional window protocols.

3 A Simple Timeout Condition

3.1 The Original Version

In the original version, the simple timeout condition is defined as follows:

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

If this condition holds, the sender process S may resend the message with sequence number na. However, while this condition ensures correct operation of the protocol, as is shown in [1], only the test $na \neq ns$ can be evaluated directly by S. Hence it is necessary to supply S with a test which implies `timeout` if it evaluates to true. The authors suggest

> ... 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.

A formal modeling of this statement in a notation which fits the rest of the protocol, is the following.

The implementation. We add to the code of S a local timer tms which is a variable of type real, with the meaning that the "timer goes off" in case $tms \leq 0$. Thus we get

in action 2: `timeout` is replaced by $(na \neq ns) \wedge (tms \leq 0)$

To set the timer tms when it sends a message (either in action 0 or in action 2), S needs a (real) constant tp, the timeout period. To implement the aging of messages in transit, we add to each message a timer field tf as first field, which is set when the message is sent. Thus both S and R need a constant mds and mdr (not necessarily the same) to set the timer field in the messages they send. These values mds and mdr can be thought of as the maximum delay of a message from S to R, and from R to S, respectively. Thus we get

in action 0:	**send** ns	is replaced by	**send** (mds, ns); $tms := tp$
in action 2:	**send** na	is replaced by	**send** (mds, na); $tms := tp$
in action 3:	**send** (v, v)	is replaced by	**send** (mdr, v, v)
in action 5:	**send** $(nr, vr - 1)$	is replaced by	**send** $(mdr, nr, vr - 1)$

To simulate the progress of time, we add an action **Time** which decreases all timers, in process(es) and messages, with the same positive amount. This can be interpreted as: during this amount of time no actions of the protocol were executed. Note that we thus assume that all timers run at exactly the same rate. This assumption is not necessary, but clarifies the exposition. The extension to timers with ρ-bounded drift is straightforward, see for example Tel [3]. To discard messages that are "too old", we test the decreased timer field in messages and delete outdated messages from the channels in the action **Time**. This now becomes as follows:

Time: **begin** choose $\delta \in \mathbb{R}^+$;
forall messages $(tf, n) \in C_{SR}$
do $tf := tf - \delta$;
if $tf \leq 0 \rightarrow$ delete (tf, n) from C_{SR} **fi**
od;
forall messages $(tf, n, m) \in C_{RS}$
do $tf := tf - \delta$;
if $tf \leq 0 \rightarrow$ delete (tf, n, m) from C_{RS} **fi**
od;
$tms := tms - \delta$
end

Note that it is not the case that the new guard of action 2 (i.e., $(na \neq ns) \wedge (tms \leq 0)$) implies `timeout`. First we have to derive what the relation is between the constants we defined. It is clear that the condition $C_{SR} = \{\}$ is implied by $tms \leq 0$ if we choose the constants tp and mds such that $tp > mds$. However, the condition $C_{RS} = \{\}$ is *not* implied by $tms \leq 0$ if we choose tp, mds, and mdr such that $tp > mds + mdr$. This is the case because although messages sent in action 3 are sent "immediately" upon receipt of a message from S (action 3 is defined as an atomic action), messages sent in action 5 may be sent an arbitrary time after the last receipt action. Thus we cannot derive any bound on the value of tp. To enable S to use a timer anyway, we demand that if R sends a message in action 5, it does so immediately upon receipt of a message from S. (Without a timer, R cannot measure any other time than "0".)

Second, there is a comparable problem with the conjunct $\neg rcvd[nr]$ of `timeout`. Its value is directly observable only for R. What we can do is to take care that this conjunct *always* holds, by formulating action 4 as a loop, always adding action 5, and doing this as soon as a new message is received in 3. The result is that the protocol for R now consists of only one atomic action:

345: **begin rcv** $v \rightarrow$ **if** $v < nr \rightarrow$ **send** (mdr, v, v)
$[\!]\ v \geq nr \rightarrow rcvd[v] := true;$
do $rcvd[vr] \rightarrow vr := vr + 1$ **od**;
if $nr < vr$
$\rightarrow$ **send** $(mdr, nr, vr - 1)$; $nr := vr$
fi
fi
end

If we now choose tp such that $tp > mds + mdr$, one can show that $(na \neq ns) \wedge (tms \leq 0)$ implies `timeout`.

The problem. The consequence of the implementation of the protocol as derived above is that R is forced to send an acknowledgment for each new message that arrives in order. Hence if almost all messages arrive in the right order (and they are sent almost all in the right order if there is a low error rate), almost all messages have to be acknowledged by a separate acknowledgment: If the "next" message (i.e., with $n = nr$) is received by R, $rcvd[nr]$ is set to $true$, (mdr, nr, nr) is sent to S, and nr increased. It is not possible to try to save on acknowledgments by sending one only for every second message that arrives, for example, as we have to take care that $\neg rcvd[nr]$ continues to hold. Thus nr has to be increased, but this is only done when an acknowledgment is sent. That it is indeed necessary for a correct operation of the protocol to demand $\neg rcvd[nr]$ follows from a slight adaptation in the counterexample provided in Subsection 4.1.

However, since the protocol is now forced to send an acknowledgment upon

almost every message, its performance will be greatly reduced, as stated by the authors themselves in discussing the protocol of Stenning [2]:

> For instance, the selective–repeat protocol in [2] requires that every data message be acknowledged by a distinct acknowledgment message. ... this is a severe restriction over the behavior of a regular window protocol, and can greatly reduce the protocol's performance.

We conclude that the suggested implementation with one timer for the sender only is not sufficient to achieve the claimed performance of the block acknowledgment protocol.

3.2 The Proposed Changes

It is quite obvious what we can do to alleviate the problem described above. The receiver R must be enabled to accumulate a number of acknowledgments in one block acknowledgment, and the sender S must wait longer for this accumulation before deciding upon a timeout. Thus we demand that if R sends an acknowledgment, it does so within a fixed time since receiving the last data message. Hence R needs a timer to measure this. We will call this timer tmr, and the constant to which it is set when R receives a message, mrt (maximum reply time). Hence we can make the sending of an acknowledgment a separate atomic action again, with an additional guard $tmr > 0$. The actions of R now are as follows:

34: **begin rcv** v $\rightarrow$ $tmr := mrt$;
if $v < nr \rightarrow$ **send** (mdr, v, v)
$[\!]$ $v \geq nr \rightarrow$ $rcvd[v] := true$;
do $rcvd[vr] \rightarrow vr := vr + 1$ **od**
fi
5: $[\!]$ $(nr < vr) \wedge (tmr > 0) \rightarrow$ **send** $(mdr, nr, vr - 1)$; $nr := vr$
end

Of course, in the action **Time** we have to decrease tmr with the same amount as all other timers, hence we add the statement "$tmr := tmr - \delta$".

It is now clear that if we choose $tp > mds + mdr + mrt$, the condition $tms \leq 0$ implies $(C_{SR} = C_{RS} = \{\}) \wedge \neg rcvd[nr]$.

4 Sophisticated Timeout Conditions

The versions of the protocol with a simple timeout condition (both the original version and our adaptation) have the drawback that if a number of consecutive data messages are lost, the retransmissions of those data messages all have to be separated by a full timeout period. Hence the authors developed a second version of the protocol with sophisticated timeout conditions to alleviate this.

4.1 The Original Version

Recall that in the first version, only the data message with sequence number na could be resent. In this version it will be possible to resend all unacknowledged data messages, i.e., with sequence numbers from na to ns. The timeout condition for a message with sequence number i is given ([1]) as:

$$\begin{aligned} \texttt{timeout}(i) \stackrel{\text{def}}{=} \; & (na \leq i \wedge i < ns \wedge \neg ackd[i]) \wedge (\#SR^i = 0) \wedge \\ & (i < nr \vee \neg rcvd[i]) \wedge (\#RS^i = 0) \end{aligned}$$

where

$\#SR^i \stackrel{\text{def}}{=}$ number of messages with sequence number i in C_{SR}

$\#RS^i \stackrel{\text{def}}{=}$ number of messages (n, m) with $n \leq i \leq m$ in C_{RS}.

Again, only the tests whether the number i lies in the right range and is not acknowledged yet can be performed directly by S, while the other conjuncts of $\texttt{timeout}(i)$ must be deduced from timer values. We quote the authors about this sophisticated timeout condition:

> ... 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.

The implementation. Instead of one timer tms, process S now has an infinite array of timers, one ($tms[i]$) for each data message. (In the version with bounded sequence numbers, this reduces to w timers for S.) Thus we get the following changes in action **Time** and in the protocol for S as given in [1]:

in **Time**:	$tms := tms - \delta$	is replaced by	**forall** $i \geq 0$ **do** $tms[i] := tms[i] - \delta$ **od**
in S:	**send** ns	is replaced by	**send** (mds, ns); $tms[ns] := tp$
	action 2	is replaced by	action 2′:

2′: $[\!] \; (na \leq i < ns) \wedge (tms[i] \leq 0) \wedge \neg ackd[i] \rightarrow$ **send** (mds, i); $tms[i] := tp$

For process R we begin by taking the original protocol consisting of the three actions 3, 4, and 5. We will now derive what timing constraints we have to impose to ensure that if the guard of action 2′ is enabled for some i, this implies condition $\texttt{timeout}(i)$. (Unfortunately, this will not be possible.) First of all, $tp > mds$ is sufficient to achieve that if the guard holds for i, $\#SR^i = 0$. For acknowledgments (i, i) sent in action 3 immediately upon receipt of a data message with number $i < nr$, we know that they have disappeared from C^{RS} within $mds + mdr$ time since the data message was sent. Hence we demand $tp > mds + mdr$. However, in the absence of any timers for R, we have no other choice than to require that R sends any acknowledgments immediately upon receipt of a data message. Thus action 5 must be included in the same atomic action as action 3, and the protocol for R now again consists of one atomic action 345, as given in Subsection 3.1.

The problem. Unfortunately, for values of i with $na < i < ns$, it is *not* the case that $\neg ackd[i] \wedge (tms[i] \leq 0)$ implies $\texttt{timeout}(i)$. The problem lies in the clause "$(i < nr) \vee \neg rcvd[i]$". This more or less requires S to "know" whether data message i arrived or not. Consider the following two scenarios which are indistinguishable for S.

S sends data messages 0, 1, 2, and 3, and receives acknowledgments (0,0) and (1,1). S resends data message 2. At this point, the variables of S have the following values: $ns = 4$, $na = 2$, $j = 1$, $i = 2$, $ackd[0] = ackd[1] = true$, $ackd[2] = ackd[3] = false$. Let S timeout for data message 3 now.

In the first scenario, process R has received data messages 0, 1, 3, and 2 (in this order), and thus has sent acknowledgments (0,0), (1,1), and (2,3). The variables of R have the following values: $nr = vr = 4$, $v = 2$, $rcvd[0] = rcvd[1] = rcvd[2] = rcvd[3] = true$. Acknowledgment (2,3) is lost. When process S timesout for data message 3, it is indeed the case that $\texttt{timeout}(3)$ holds, as $3 < nr = 4$.

In the second scenario, process R has received data messages 0, 1, and 3, data message 2 is lost, and thus R has sent acknowledgments (0,0) and (1,1). The variables of R have the following values: $nr = vr = 2$, $v = 3$, $rcvd[0] = rcvd[1] = rcvd[3] = true$, and $rcvd[2] = false$. When process S timesout for data message 3, it is *not* the case that $\texttt{timeout}(3)$ holds, as $3 \not< 2$ and $rcvd[3] = true$. Thus the second clause of assertion 8 ([1]): $\forall m : {}^{\#}SR^{m} > 0 : m < ns \wedge \neg ackd[m] \wedge (m < nr \vee \neg rcvd[m])$ is violated for $m = 3$. Assertion 8 is part of the invariant which implies the safety of the block acknowledgment protocol (see Section 2).

Is it possible to prevent that a situation like this, i.e., that $tms[i] < 0$ while $\neg\texttt{timeout}(i)$, occurs in the protocol? The problem lies in the timeout condition for the data message with number i in case S has not received an acknowledgment yet for messages numbered $i-1$ and higher. The condition $i < nr \vee \neg rcvd[i]$ requires S to decide whether the acknowledgment $(i-1, i)$ was lost (timeout warranted) or data message $i-1$ was lost while data message i was received (timeout *not* warranted).

One trivial solution is to prevent that message i is sent if message $i-1$ is not acknowledged yet. This amounts to setting $w = 1$, and reduces to the alternating bit protocol. This is not the protocol meant by the authors. The second trivial solution is to prevent that message i is resent if message $i-1$ is not acknowledged yet. This amounts to the protocol with simple timeout conditions, and is not the protocol meant by the authors.

We cannot force R to send an acknowledgment for i if $i-1$ is not received yet, as this is not in accordance with the specification of the protocol. We began our attempt to implement the protocol with sophisticated timeouts with timers in S only, but would it help to use any additional timers? The only use we can make of timers is to prohibit the execution of an otherwise enabled action subject to a certain condition on the value of a timer. However, both situations (all acknowledgments of i and $i-1$ lost; and i received but all (re)transmissions of $i-1$ lost, respectively) can exist for an arbitrary time (i.e., finite but unbounded). Note that the assumption of *fairness* only states that messages that are sent infinitely often, eventually will be

received, not that this will happen within some bounded time. We thus conclude that an implementation of the block acknowledgment protocol with sophisticated timeout conditions is *not* possible by means of timers. Hence we do not consider it correct.

An Erroneous Execution. Above we have only shown that in the second scenario, it is possible that the invariant of the protocol which implies its safety, is violated. As such, this does not necessarily mean that the protocol operates incorrectly. In the versions of the protocol that we discussed so far, we used unbounded sequence numbers, and hence each message is uniquely identified by its number. It is only when we convert the protocol to a version which uses finite sequence numbers, that we can pin–point an erroneous execution where one message is mistaken for another. Hence we will proceed to show that this can actually happen if the second scenario as sketched above occurs.

We define the remaining constants as follows. Let $mds = mdr = 1$ and $tp = 3$. Let $w = 2$ and $n = 4$. It is shown in [1] that it is possible to use sequence numbers and to do all computations mod n when $n \geq 2w$. In process R, the sequence number v of a received data message is expected to lie between $nr - w$ (inclusive) and $nr + w$ (exclusive). Thus we show an execution in Figure 1 where the illegal retransmission of data message 3 is considered as a receipt of data message 7 by R ($3 = 7 \bmod n$). This will be the case if $nr - w = 4$ and thus if $nr = 6$. As a consequence of the illegal timeout, data message 3 is on its way to R, while in the mean time, when R receives the also resent data message 2, acknowledgment (2,3) is sent to S. If it is received, na is increased to 4 and S sends the next two data messages 4 and 5, with as sequence numbers 0 and 1. These arrive fast at R, before the resent message 3. As data message 3 had already been received by R before, R increments nr to 5 and 6 upon receipt of messages 0 (i.e., 4) and 1 (i.e., 5). When the resent data message 3 is received by R, nr has the value of 6, and as R expects that the sequence number of a received message lies between $nr - w$ and $nr + w$, that is, between 4 and 8 in this case, R considers this to be the message with sequence number 7. Thus this execution with bounded sequence numbers is erroneous.

We conclude that it is indeed *necessary* to keep the assertion $\forall m :: (\#SR^m + \#RS^m) \leq 1$ always true.

4.2 The Proposed Changes

As we showed above that it is not observable for S whether $i < nr \vee \neg rcvd[i]$ holds in R, we will have to ensure that a situation as sketched above cannot occur any more. Thus we will now ensure that R does not send an acknowledgment for a data message that might be retransmitted by S because its timer goes off. Hence we again introduce a "maximum reply time", now for each data message separately, and only allow R to send one block acknowledgment for several data messages at

"time"	S		channels	R	
	action	na		receipt	nr
0	0: **send** 0	0		**rcv** 0	1
	0: **send** 1	0		**rcv** 1	2
	1: **rcv** $(0,0)$	1			
Time with $\delta = 1$					
1	0: **send** 2	1	lost		
	1: **rcv** $(1,1)$	2			
Time with $\delta = 1$					
2	0: **send** 3	2		**rcv** 3	2
Time with $\delta = 2$					
4	$2'$: **send** 2	2		**rcv** 2	4
Time with $\delta = 1$					
5	$2'$: **send** 3	2			
	1: **rcv** $(2,3)$	4			
	0: **send** 0 $(= 4)$	4		**rcv** 0 $(= 4)$	5
	0: **send** 1 $(= 5)$	4		**rcv** 1 $(= 5)$	6
				rcv 3 $(\neq 7)$	6 $\Longleftarrow$
Time with $\delta = 1$					
6	1: **rcv** (0,0)	5			

Executions of actions are ordered from top to bottom, and within one line from left to right.

Figure 1: An Erroneous Execution.

once if the maximum reply time is not exceeded for any of them. Hence we supply R with a timer for each possible data message.

As for the possibility to use only one timer in R instead of one for each data message, this leads to an unexceptable performance. (We do not consider the possibility to simulate n timers by one timer, which is of course possible.) If R has only one timer, it can only measure the time since it received a specific message. For this specific message, one has the choice between two possibilities.

First, one could choose the last message, as in the protocol with simple timeouts. To achieve a correct protocol one then has to restrict either R to always sending an acknowledgment for one data message at a time, or to restrict S to timeouts for data message na only (i.e., the protocol with simple timeouts). Hence we do not get a protocol with a performance as claimed by the authors.

Second, one could choose the timer to refer to the receipt of a data message of a certain number, such as e.g. nr or $nr + 1$. The latter number is the one that would have prevented the erroneous execution of Figure 1. However, while a timer for the

receipt of data message $nr+1$ makes it possible to send one acknowledgment $(m, m+1)$ if m and $m+1$ arrive out of order, it then forces R to send two acknowledgments (m, m) and $(m+1, m+1)$ if they arrive in order. Hence it would probably be the best choice to set the timer for the receipt of data message nr, to enable R to send one acknowledgment for two consecutive data messages at once, (assuming that the probability that messages arrive in order is higher than the probability that they arrive out of order). But this still does not give R the desired flexibility in sending acknowledgments.

Hence we assume that process R has an infinite array of timers tmr. (This reduces to $2w$ timers in the version with bounded sequence numbers.) The timer $tmr[i]$ is used to measure whether the maximum reply time mrt has elapsed or not since R received (the last occurrence of) data message i, in order to decide whether R can send an acknowledgment for i or not. Thus $tmr[i]$ has to be set upon receipt of data message i and action 3′ of R becomes as follows:

3′: **rcv** $v \rightarrow tmr[v] := mrt;$
if $v < nr \rightarrow$ **send** (mdr, v, v)
$[\!] \; v \geq nr \rightarrow rcvd[v] := true$
fi

We have to choose the timeout period tp in S accordingly, such that $tp > mds + mdr + mrt$. As we now have to check all timer values $tmr[i]$ of those values i that are candidates to be included in a block acknowledgment sent in action 5, we include action 4 in the new action 45′, subject to the values in the relevant timers. (As a consequence nr always equals vr outside an action.) Thus we have to change the guard of the action to the condition that ensures that vr will indeed be increased inside the new action 45′. This condition is $rcvd[nr] \wedge tmr[nr] > 0$. Hence action 45′ of R becomes as follows:

45′: $rcvd[nr] \wedge tmr[nr] > 0 \rightarrow$ **do** $rcvd[vr] \wedge tmr[vr] > 0 \rightarrow vr := vr + 1$ **od**;
send $(mdr, nr, vr - 1)$; $nr := vr$

Of course, in the action **Time** we have to decrease $tmr[i]$ with the same amount as all other timers, hence we add the statement "**forall** $i \geq 0$ **do** $tmr[i] := tmr[i] - \delta$ **od**".

For ease of reference, we now give the complete code of the protocol as derived above.

process S;
const w : **integer val**; /*$w > 0$*/
tp, mds : **real val**; /*$tp > mds + mdr + mrt$, $mds \geq 0$*/
var $ackd$: **array[integer] of Boolean init** *false*;
na, ns, i, j : **integer init** 0;
tms : **array[integer] of Timer** /***real***/ **init** 0;
0: **begin** $ns < na + w \rightarrow$ **send** (mds, ns); $tms[ns] := tp$;

$ns := ns + 1$

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

2′: $[]\ (na \leq i < ns) \wedge (tms[i] \leq 0) \wedge \neg ackd[i] \rightarrow$ **send** $(mds, i);\ tms[i] := tp$

end

process R;

const mrt, mdr : **real val** /*≥ 0*/

var $rcvd$: **array[integer] of Boolean init** *false*;

nr, vr, v : **integer init** 0;

tmr : **array[integer] of Timer** /***real***/ **init** 0;

3′: **begin rcv** v $\rightarrow tmr[v] := mrt;$
if $v < nr \rightarrow$ **send** (mdr, v, v)
$[]\ v \geq nr \rightarrow rcvd[v] := true$
fi

45′: $[]\ rcvd[nr] \wedge tmr[nr] > 0 \rightarrow$ **do** $rcvd[vr] \wedge tmr[vr] > 0 \rightarrow vr := vr + 1$ **od**;
send $(mdr, nr, vr - 1);\ nr := vr$

end

Time;

var δ : **real**;

begin choose $\delta \in \mathbb{R}^+$;
forall messages $(tf, n) \in C_{SR}$
do $tf := tf - \delta$;
if $tf \leq 0 \rightarrow$ delete (tf, n) from C_{SR} **fi**
od;
forall messages $(tf, n, m) \in C_{RS}$
do $tf := tf - \delta$;
if $tf \leq 0 \rightarrow$ delete (tf, n, m) from C_{RS} **fi**
od;
forall $i \geq 0$ **do** $tms[i] := tms[i] - \delta$ **od**;
forall $i \geq 0$ **do** $tmr[i] := tmr[i] - \delta$ **od**

end

Loss;

var C_{SR}, C_{RS} : **multiset of messages init** $\{\}$;

begin choose a message $\in C_{SR} \cup C_{RS}$; delete it

end

5 Correctness Proof

Although we now have formulated a correct protocol as we will show below, it is not a strict implementation of the protocol with sophisticated timeouts from [1] as our timeout condition for action 2': $(na \leq i < ns) \wedge (tms[i] \leq 0)$ does *not* imply $\texttt{timeout}(i)$. It is however based on the same principle, namely to keep the assertion $\forall m :: (\#SR^m + \#RS^m) \leq 1$ always true. If we consider again the erroneous execution in Subsection 4.1, S still "illegally" timesout for data message 3 (as $(3 \not< nr) \wedge rcvd[3]$), but the acknowledgment (2,3) would not have been sent (in the execution of Figure 1, as $tp = 3$, mrt must be choosen < 1 and thus $tmr[3] < -1$ at the moment R receives data message 2 and considers sending acknowledgment $(2,3)$), and hence $\#SR^m + \#RS^m$ remains ≤ 1 for $m = 3$.

5.1 Safety

In view of the fact that the changed protocol is not a strict implementation of the original version, it perhaps not surprising that we need a slightly different invariant to prove the safety of the changed protocol. In [1], this invariant is the conjunct of three assertions 6, 7, and 8. Of these three, we can slightly strengthen assertion 6 to 6', while assertion 7 remains the same:

6': $na \leq nr = vr \leq ns \leq na + w$
7: $(\forall m : \neg ackd[m] : m \geq na) \wedge (\forall m : ackd[m] : m < nr) \wedge \neg ackd[na] \wedge$
$(\forall m : rcvd[m] : m < ns) \wedge (\forall m : \neg rcvd[m] : m \geq vr)$

We have to weaken assertion 8 to 8', because in assertion 8 the clause $m < nr \vee \neg rcvd[m]$ which ensures that R will not send an acknowledgment for data message m, does not hold any more. It is replaced by $tmr[m] \leq 0$ in assertion 8', which has the same effect.

8': $(\forall m :: (\#SR^m + \#RS^m) \leq 1) \wedge$
$(\forall m : \#SR^m > 0 : m < ns \wedge \neg ackd[m] \wedge tmr[m] \leq 0) \wedge$
$(\forall m : \#RS^m > 0 : m < nr \wedge \neg ackd[m])$

For a safe transition to bounded sequence numbers and finite arrays, we need assertion 9, which is invariant in conjunction with the other assertions. This assertion again holds for both the original and the changed protocol.

9: $(\forall m : \#SR^m > 0 : nr - w \leq m < nr + w) \wedge$
$(\forall m : \#RS^m > 0 : na \leq m < na + w)$

For the proof of the safety of the changed protocol we need an additional assertion (10) that relates the values of the timers in the protocol to the other variables. Let tf^m denote the value of the timer field of the message in transit concerning m if it exists. We know with assertion 8' that there is at most one message concerning m.

10: $(\forall m : tms[m] \leq 0 : (\#SR^m = 0) \wedge (tmr[m] \leq 0) \wedge (\#RS^m = 0)) \wedge$
$(\forall m : \#SR^m > 0 : tms[m] > mrt + mdr + tf^m) \wedge$
$(\forall m : tmr[m] > 0 : (\#SR^m = 0) \wedge (tms[m] > mdr + tmr[m])) \wedge$
$(\forall m : \#RS^m > 0 : tms[m] > tf^m)$

For the proof of the safety of the changed protocol, one can easily check that the conjunction of assertions 6′–10 is invariant under all possible actions of the system, that is, actions 0, 1, 2′ of S, actions 3′ and 45′ of R, action **Time**, and action **Loss**. The latter deletes a message from a channel. As channels are modeled as multisets of messages, we do not need to model the reordering of messages by a separate action. Hence the proof of fault tolerance is included in the proof of safety.

5.2 Progress

To prove progress, it is sufficient to show that na is incremented infinitely often. This is the case because nr and ns remain within a distance w of na by assertion 6′. First consider the case that there are no losses of messages. We then have the following sequence of enabled actions whose execution enables other actions:

<table>
<tr><td>enabled action</td><td colspan="2">$0(na) \vee 2'(na) \vee$ Time; $2'(na)$</td></tr>
<tr><td>execution results in</td><td colspan="2">$\#SR^{na} = 1$</td></tr>
<tr><td>enabled action</td><td colspan="2">$3'(na)$</td></tr>
<tr><td>execution results in</td><td>case 1: $na = nr$
$tmr[nr] > 0$</td><td>case 2: $na < nr$
$\#RS^{na} = 1$</td></tr>
<tr><td>enabled action</td><td>$45'(na)$</td><td rowspan="2">⋮
⋮</td></tr>
<tr><td>execution results in</td><td>$\#RS^{na} = 1$</td></tr>
<tr><td>enabled action</td><td colspan="2">$1(na)$</td></tr>
<tr><td>execution results in</td><td colspan="2">$na := na + 1$</td></tr>
</table>

Note that the sending of data message na is enabled infinitely often. By actions **Loss** and **Time** the actions 3′ and 1 can be disabled, but with the assumption of *fairness* they will be eventually executed, and na increased. Thus na is increased infinitely often. As we know by assertions 6′ and 7 that all messages $< na$ are received by R, infinitely many (new) messages are received.

References

[1] BROWN, G. M., M. G. GOUDA, AND R. E. MILLER. Block acknowledgment: redesigning the window protocol. *IEEE Trans. on Commun.*, vol. 39-4, 524-532, 1991.

[2] STENNING, N. V. A data transfer protocol. *Computer Networks*, vol. 1, 99-110, 1976.

[3] TEL, G. *Topics in Distributed Algorithms*, vol. 1 of *Cambridge International Series on Parallel Computation*. Cambridge University Press, Cambridge, U.K., 1991.