Verification of Connection-Management Protocols

Anneke A. Schoone

RUU–CS–87–14
August 1987

Verification of Connection-Management Protocols

Anneke A. Schoone

Technical Report RUU-CS-87-14
August 1987

Department of Computer Science
University of Utrecht
P.O. Box 80.012, 3508 TA Utrecht
the Netherlands.

VERIFICATION OF CONNECTION-MANAGEMENT PROTOCOLS

Anneke A. Schoone

Department of Computer Science, University of Utrecht,
P.O.Box 80.012, 3508 TA Utrecht, the Netherlands.

Abstract. It was informally shown by Belsnes that k-way handshakes for connection management can be reliable only if $k \geq 4$, even in the case that a single message has to be transmitted reliably during each connection. We give a rigid proof of this fact using the Krogdahl-Knuth technique of system-wide invariants. The proof leads to a better insight into the subtleties of connection management, resulting in several shorter handshakes which are reliable in slightly different models of communication.

1. Introduction.

Consider a communication network in which processors want to transmit many short but independent messages to each other. A processor can incorporate such a message in a packet and send the packet over to the destination processor. However, the communication network can loose packets, delay packets arbitrarily long, and deliver packets in a different order than the order in which they were sent. Clearly a packet is not delivered before it is sent.

We consider the problem of designing protocols that handle the communication of messages correctly, in the sense that there is no loss or duplication of messages (cf. Belsnes [1]). To specify this more precisely, suppose processor P wants to transmit a message m to Q. m is said to be lost if P thinks that Q received m while this is not the case. m is said to be duplicated if Q receives two copies of m from P and thinks they are different messages. If a processor P has a message or a sequence of messages to send to Q, it sets up a temporary connection with Q, which is closed down as soon as P knows that Q received the message(s) (or that Q is not in a position to receive them). If only a single message is transmitted during every connection, we talk about single-message communication. Note that this does not mean that only a single packet is sent over, as we might very well use a so-called k-way handshake, i.e., an exchange of k different packets between two processors. If a sequence of messages is sent over during one connection, we talk about multiple-message communication. It is assumed that it is not feasible to maintain any information about previous connections. As a consequence, it is easier to prevent loss and duplication within one and the same connection, than to do so between connections. By restricting ourselves for the moment to single-message communication, the latter problem reduces to the problem of connection management.

Belsnes [1] investigated k-way handshake protocols for connection management. He showed that it is impossible to have a protocol which ensures correct communication under the assumptions stated above, when processors can loose their information about current connections, e.g. because of a crash. He showed also that in the absence of processor failures there can be circumstances in which his 1-way, 2-way, and 3-way handshakes can lead to loss or duplication. Belsnes [1] gave a 4-way and a 5-way handshake for which he showed informally that they communicate messages correctly.

In this paper we give a rigid proof that any 1-way, 2-way, or 3-way handshake must be liable to incorrect communication, even in the absence of processor failures. To do this, we introduce a so-called protocol skeleton which describes only those protocol features that are concerned with connection management. From this we obtain all the protocols Belsnes considered in [1] by setting certain parameters and prescribing a certain order of operations. The protocol skeleton is analyzed with system-wide invariants (Krogdahl [4], Knuth [3]), and builds on the work done in [5]. Furthermore, we give a class of k-way handshakes (for any $k \geq 4$) which ensure correct communication in the absence of processor failures. The connection management protocols are then extended to ensure correct multiple-message communication. We proceed by analyzing the necessary changes in the model of communication to achieve reliable communication with shorter handshakes.

The paper is organized as follows. The protocol skeleton will be presented in section 2, while the mathematical analysis can be found in section 3. Loss and duplication will be formally defined in section 4. Since Belsnes' protocols depend critically on the use of error packets, we will deal with them in section 5. The selection of certain parameters will be discussed in section 6, resulting in a proof that four-way handshakes are required to achieve reliable communication of messages. In section 7 it is shown how the basic protocol skeleton can be extended to handle multiple-message communication. Finally, in section 8 we study what the effect is on the correctness of the communication, if we weaken several assumptions in the model. This leads to several shorter handshakes which are reliable for the resulting weaker models of communication.

2. A protocol skeleton. Careful consideration of the k-way handshakes Belsnes [1] gives for reliable single-message communication leads to the insight that the short packet exchanges basicly follow a sliding-window protocol with window size equal to one (see [5]). That is, inside one connection, a processor sends a packet of a certain type (number), and after receipt of a packet of the expected type (number), it sends a packet of the next type (number). Note, however, that the results of [5] do not carry over directly, since we do not make the assumption here that the sending order of packets is preserved. On the other hand, we need not be concerned now about sequence numbers wrapping around inside one connection, since we only consider k-way handshakes with k small.

Our main concern is the identification of connections, to prevent confusion with earlier connections and their packets. A connection between two processors P and Q is identified (in accordance with Belsnes) by means of two identifiers which originate at P and Q respectively. For this purpose, each processor has a function '*new value*' which produces a new, unique value unequal zero each time it is called. Hence the value produced by a call to '*new value*' is different from all previous values and unguessable for any other processor. The connection management relies heavily on the availability of such a function. (We do not go into the problem of finding such a function here. See e.g. Tomlinson [7].)

Although in general the communication network contains many processors and many pairs of processors setting up and closing connections, we restrict our attention to only one pair of processors which repeatedly want to communicate. Hence we ignore the control information in the packets pertaining to the processor identities and assume that the packets concerning this one pair of processors are filtered out correctly. For the moment we are not interested which packet contains the actual message either, and thus restrict our attention to the three control fields used for the actual connection management. For our purpose packets thus have the following form: $<mci,yci,seq>$, where mci is 'my connection identifier' (i.e. the connection identifier used by the sender of the packet), yci is 'your connection identifier' (the connection identifier used by the receiver in case of a reply), and seq is the sequence number of the packet within the connection.

Except that a processor needs to remember mci and yci during a connection, it needs a constant g and a variable a (local to the connection), see [5]. Since we restrict our attention to the connections between one pair of processors P and Q, we will only add P and Q as subscripts to denote the processor to which a variable or operation pertains. The protocol skeleton gives P the possibility to do the atomic operations S_P (send), R_P (receive) and C_P (close) in any order and as often as desired. (It is understood that a close operation has no effect if the connection is already closed.) Both the send and receive operation provide for the opening of a (new) connection as required.

```
$S_P$:  if connection with Q closed then      (*open*)
        begin $mci_P$:=$new\ value$; $yci_P$:=0; $g_P$:=1; $a_P$:=0 end;
        send a packet $<mci_P,yci_P,seq>$ to Q where $0 \le seq < a_P + g_P$.

$R_P$:  receive $<x,y,z>$;
        if connection with Q closed
        then  if $z \ne 0$ then $error$ $(<x,y,z>)$
              else begin $mci_P$:=$new\ value$; $yci_P$:=$x$; $g_P$:=0; $a_P$:=1 end      (*open*)
        else  if not (($x=yci_P$ or $a_P=0$)      (*x valid*)
                  and $y=mci_P$                  (*y valid*)
                  and $z=a_P$)                   (*z valid*)
              then $error$ $(<x,y,z>)$
              else begin $a_P$:=$a_P$+1; if $z$=0 then $yci_P$:=$x$ end

$C_P$:  if connection with Q is open then      (*close*)
        begin if $a_P \le cf_P$ then report failure;
              if $a_P \ge cs_P$ then report success;
              $mci_P$:=undefined; $yci_P$:=undefined; $a_P$:=undefined; $g_P$:=undefined
        end
```

where

a_P	the number of packets that P has received from Q during this connection. Thus $a_P \ge 0$.
g_P	a parameter which encodes the direction of data transfer. g_P=1 if P decided to send Q some information, and g_P=0 if P opened the connection on receipt of a packet from Q.
mci_P	my connection identifier.
yci_P	your connection identifier, which P has copied from the packet it received from Q. In

case P has received no packet yet, $yci_P=0$.

cf_P and cs_P are values, only depending on g_P, which are used to decide whether the connection being closed was a failure (no message came across), or a success (the message came across). Necessarily $cs_P>cf_P$.

error is a procedure which could be 'ignore' or 'send an error packet'. We will discuss what it should do in section 5. For the moment it suffices to know that it does not change any of the variables.

new value produces a new unique value ($\neq 0$) each time it is called, to serve as an identifier of the new connection.

$valid_P(<x,y,z>)$ is a shorthand notation for the test whether the packet $<x,y,z>$ should be considered in the operation R_P. (It could be a retransmission of some old packet.) Hence it is defined as

```
valid_P(<x,y,z>)= if connection with Q is closed then z=0
                  else (x=yci_P or a_P=0)   (*x valid*)
                       and y=mci_P          (*y valid*)
                       and z=a_P)           (*z valid*)
```

Sometimes we will need to specify the time when a variable has a certain value: thus $a_P(t)$ will denote the value (possibly undefined) of the variable a_P at time t.

3. Invariants.

Since P and Q use the same operations S, R and C, lemmas that hold for P hold for Q also, with P and Q interchanged. We will only state and prove them for one processor. In order to formulate the invariants, we need some predicates to express events like 'a packet was sent', 'a packet was received', 'a packet was accepted as valid', and 'a connection was closed', respectively.

Definition 3.1. An (m,y,g,a) close for P is an operation C_P, invoked at some time t by P, such that $mci_P(t)=m$, $yci_P(t)=y$, $g_P(t)=g$ and $a_P(t)=a$ (We consider only meaningful closes, i.e. $m\neq$undefined). The predicate $closed_P(m,y,g,a)$ becomes true when P does an (m,y,g,a) close.

The predicate $sent_P(<x,y,z>)$ becomes true when P does an operation S_P in which it sends a packet $<x,y,z>$.

The predicate $received_P(<x,y,z>)$ becomes true when P does an operation R_P in which it receives a packet $<x,y,z>$.

The predicate $accepted_P(<x,y,z>)$ becomes true when P does an operation R_P in which the received packet $<x,y,z>$ is accepted as valid.

Thus for example, $closed_P(m,y,g,a)(t)$ is true when P did the (m,y,g,a) close at or before time t.

Lemma 3.1.

(1) $closed_P(m,y,g,a) \Rightarrow mci_P\neq m$.

(2) Let $t_1\leq t_2$ and $mci_P(t_1)\neq$undefined. Then

(i) $mci_P(t_1)=mci_P(t_2) \Leftrightarrow \neg\, closed_P(mci_P(t_1),y,g,a)(t_2)$.

(ii) $mci_P(t_1) = mci_P(t_2) \Rightarrow \quad ((g_P(t_1) = g_P(t_2) \wedge a_P(t_1) \le a_P(t_2) \wedge$
$(yci_P(t_1) = yci_P(t_2) \vee (yci_P(t_1) = a_P(t_1) = 0 \wedge g_P(t_1) = 1)))$.

Proof. Obvious from the protocol skeleton. ■

Lemma 3.2.

(1) $received_P(<x,y,z>) \Rightarrow sent_Q(<x,y,z>)$.

(2) $accepted_P(<x,y,z>) \Rightarrow received_P(<x,y,z>)$.

(3) $sent_P(<x,y,z>) \Rightarrow ((x = mci_P \wedge (y = yci_P \vee (y = z = 0 \wedge g_P = 1)) \wedge z < a_P + g_P) \vee$
$(closed_P(x, y', g', a') \wedge (y = y' \vee (y = z = 0 \wedge g' = 1)) \wedge z < a' + g'))$.

(4) $accepted_P(<x,y,z>) \Rightarrow ((x = yci_P \wedge (y = mci_P \vee z = g_P = 0) \wedge z < a_P) \vee$
$(closed_P(m', x, g', a') \wedge (y = m' \vee z = g' = 0) \wedge z < a'))$.

Proof. (1). We do not allow that packets are received which were not sent. Moreover, we assume that addressing is done correctly.
(2). Obvious.
(3). $sent_P(<x,y,z>)$ becomes true when P does an operation S_P and sends $<x,y,z>$. Hence at that moment $x = mci_P$, $y = yci_P$, and $z < a_P + g_P$. If $y = 0$, then $g_P = 1$ and $a_P = 0$, hence $z = 0$. R_P can increase a_P, which leaves $z < a_P + g_P$ valid, and change yci_P, but only in case a_P was 0 before, thus $y = z = 0 \wedge g_P = 1$ holds now. C_P invalidates $x = mci_P$ but now $closed_P(x, y', g', a') \wedge (y' = y \vee (y = z = 0 \wedge g' = 1)) \wedge z < a' + g'$ holds.
(4). $accepted_P(<x,y,z>)$ becomes true when P does an operation R_P in which $<x,y,z>$ is valid and is accepted. There are two cases.
Case 1. P was closed. Thus $z = 0$ and after the opening of the connection by P we have $yci_P = x$, $g_P = 0$ and $a_P = 1$. Hence the relation holds.
Case 2. P was open. Thus $y = mci_P$, $z = a_P + 1$, $x = yci_P$ at the completion of R_P. Hence the relation holds.
Another operation R_P during the same connection can only increase a_P, which keeps the relation valid. Operations S_P, S_Q, C_Q and R_Q do not change any of the variables involved. Finally, when P does an operation C_P to close this connection, $closed_P(m', x, g', a') \wedge (y = m' \vee (z = g' = 0)) \wedge z < a'$ becomes true. This cannot be invalidated anymore by any of the operations. ■

Lemma 3.3.

(1) $a_P \ge 1 \Leftrightarrow (yci_P \ne 0 \wedge yci_P \ne \text{undefined})$.

(2) $a_P \ge 1 \Rightarrow (accepted_P(<yci_P, y, a_P - 1>) \wedge (y = mci_P \vee a_P - 1 = g_P = 0))$.

(3) $a_P \ge 1 \Rightarrow ((mci_Q = yci_P \wedge g_Q + g_P \le 1 \wedge a_P \le a_Q + g_Q) \vee$
$(closed_Q(yci_P, y, g, a) \wedge g + g_P \le 1 \wedge a_P \le a + g))$.

(4) $(closed_P(m', y', g', a') \wedge a' \ge 1) \Rightarrow$
$((mci_Q = y' \wedge g_Q + g' \le 1 \wedge a' \le a_Q + g_Q) \vee (closed_Q(y', y, g, a) \wedge g + g' \le 1 \wedge a' \le a + g))$.

Proof. (1). Obvious from the protocol skeleton.
(2). Obvious from the protocol skeleton.
(3). Combining lemmas 3.3(2) and 3.2 directly leads to the desired result if we note that the cases $g_P + g_Q = 2$ and $g + g_P = 2$, respectively, lead to a contradiction.

(4). The operation C_P which invalidates $a_P \geq 1$, leads to relation (4). All other operations leave the relation invariant for the same reasons as in (3). ■

Lemma 3.4.

(1) $(mci_P = yci_Q \neq \text{undefined} \wedge mci_Q = yci_P \neq \text{undefined}) \Rightarrow g_P + g_Q = 1$.

(2) $closed_P(yci_Q, mci_Q, g, a) \Rightarrow g + g_Q = 1$.

Proof. (1). Initially the relation holds because P and Q are closed. S_P keeps the relation invariant, because if P already had an open connection with Q, S_P does not change any variables, and if P was closed, P puts $mci_P := \textit{new value}$, hence $mci_P = yci_Q$ cannot hold yet. Thus S_P keeps (1) invariant, as does S_Q. C_P and C_Q keep (1) invariant because mci_P and mci_Q, respectively are put to undefined. R_P keeps (1) invariant: If P had an open connection with Q already, g_P nor g_Q is changed, hence the sum stays the same. However, yci_P might change to some value x, such that $mci_P = yci_Q \wedge yci_P = mci_Q$ now holds. If it does, the packet received was $<x, y, z>$ with $y = mci_P$, $z = a_P = 0$, after which a_P was set to 1. Hence $g_P = 1$. Thus, if $mci_Q = yci_P$, $g_P + g_Q \leq 1$ (lemma 3.3). Hence $g_Q = 0$ and $g_P + g_Q = 1$. If R_P opens the connection with Q, it puts $mci_P := \textit{new value}$, thus $mci_P = yci_Q$ cannot hold yet. Likewise, R_Q keeps the relation invariant.

(2). If operation C_P makes $closed_P(yci_Q, mci_Q, g, a)$ true, we know with (1) that the relation holds. S_P and R_P do not change any variables, nor do S_Q and R_Q. C_Q invalidates $closed_P(yci_Q, mci_Q, g, a)$. ■

Lemma 3.5.

(1) $a_P + g_P \geq 2 \Rightarrow accepted_P(<yci_P, mci_P, a_P - 1>)$.

(2) $a_P + g_P \geq 2 \Rightarrow ((mci_Q = yci_P \wedge yci_Q = mci_P \wedge g_P + g_Q = 1 \wedge a_P \leq a_Q + g_Q \wedge a_Q \leq a_P + g_P) \vee$
$(closed_Q(yci_P, mci_P, 1 - g_P, a) \wedge a_P \leq a + 1 - g_P \wedge a \leq a_P + g_P))$.

(3) $(closed_P(m, y, g, a) \wedge a + g \geq 2) \Rightarrow$
$((mci_Q = y \wedge yci_Q = m \wedge g + g_Q = 1 \wedge a \leq a_Q + g_Q \wedge a_Q \leq a + g) \vee$
$(closed_Q(y, m, 1 - g, a') \wedge a \leq a' + 1 - g \wedge a' \leq a + g))$.

Proof. (1). Follows from lemma 3.3 and the fact that we can exclude the case $a_P - 1 = g_P = 0$.
(2). Follows from the previous lemmas if we note that for example in the case that Q is still open with $mci_Q = yci_P$, $a_Q \geq 1$, and thus lemma 3.3 can be used for Q.
(3). Follows from lemma 3.5 (2). ■

Lemma 3.6.

(1) $closed_P(yci_Q, mci_Q, g, a) \Rightarrow (\neg(a_P + g_P \geq 2) \wedge \neg closed_Q(m', mci_P, g', a'))$.

(2) $(a_P + g_P \geq 2 \wedge a_Q + g_Q \geq 2) \Rightarrow (mci_P = yci_Q \wedge yci_P = mci_Q)$.

Proof. (1). $closed_P(yci_Q, mci_Q, g, a)$ implies $mci_Q \neq \text{undefined}$, $a_Q \geq 1$, $g + g_Q = 1$, and $a \geq 1$. If P is closed, $\neg closed_Q(m', mci_P, g', a')$ holds. Initially the relation holds. If R_Q or S_Q opens a connection (and sets mci_Q), $closed_P(yci_Q, mci_Q, g, a)$ can not hold yet. It can become true in two ways: firstly, by an operation C_P if $mci_P = yci_Q$ and $yci_P = mci_Q$. Here after a_P and g_P are undefined, hence $a_P + g_P \geq 2$ does not hold. Secondly, by an operation R_Q which sets yci_Q to m while $closed_P(m, mci_Q, g, a)$ did hold already. Hence $g_Q = 1$. However, after the C_P operation which led to $closed_P(m, mci_Q, g, a)$, $\neg(a_P + g_P \geq 2)$ did hold. The only way

$a_P+g_P \geq 2$ can become true is by an R_P operation in which the packet $<yci_P, mci_P, 1-g_P>$ is accepted. However, since we had $\neg(closed_Q(m', mci_P, g', a')$ when the connection was opened by P, and Q either has $yci_Q=0$ or $yci_Q=m \neq mci_P$, $sent_Q(<m', mci_P, z>)$ does not hold. Thus $\neg(a_P+g_P \geq 2)$ as long as Q is open with the current connection. The only way $closed_Q(m', mci_P, g', a')$ can become true is by an operation C_Q, however C_Q invalidates $closed_P(yci_Q, mci_Q, g, a)$.
(2). Use lemma 3.5 (2) and 3.6 (1). ■

We are now ready to state the invariants that relate closes of P to closes of Q.

Lemma 3.7. Let $closed_P(m, y, g, a)$ be true. Then

(1) $closed_Q(y, m, g', a') \Rightarrow (g+g'=1 \wedge (a'=a+g \vee a'=a+g-1))$.

(2) $(closed_Q(m', m, g', a') \wedge m' \neq y) \Rightarrow (g'=0 \wedge a'=1)$.

(3) $(closed_P(m'', y, g'', a'') \wedge m'' \neq m) \Rightarrow$
$$((a=1 \wedge g=0) \vee (a''=1 \wedge g''=0) \vee (y=a=a''=0 \wedge g=g''=1)).$$

(4) If connections are always closed eventually and Q never does an (m', y', g', a') close with $y'=m$, then $a+g=1$.

(5) If connections are always closed eventually and Q never does an (m', y', g', a') close with $y=m'$, then $a=0$, $g=1$ and $y=0$.

Proof. (1). $m \neq 0$ and $y \neq 0$ imply $a \geq 1$ and $a' \geq 1$. From lemma 3.4 it easily follows that $g+g'=1$. Hence $a+g \geq 2$ or $a'+g' \geq 2$. Thus we can use lemma 3.5 for either P or Q with the desired result.
(2). Since $m \neq 0$, $a' \geq 1$. We know $a'+g' \geq 2$ would imply $m'=y$, thus $a'=1$ and $g'=0$.
(3). Assume $a+g \geq 2$ and $a''+g'' \geq 2$. Then with lemma 3.5 we have $(mci_Q=y \wedge yci_Q=m) \vee closed_Q(y, m, 1-g', a')$ and $(mci_Q=y \wedge yci_Q=m'') \vee closed_Q(y, m'', 1-g', a')$. Contradiction. Hence $a+g=1$ or $a''+g''=1$. Now if $y=0$ then $a=a''=0$ and $g=g''=1$. If $y \neq 0$, $a \geq 1$ and $a'' \geq 1$. Thus either $g=0 \wedge a=1$ or $g''=0 \wedge a''=1$.
(4). For $a+g \geq 2$, lemma 3.5 tells us that Q must have been open with $yci_Q=m$, hence we have a contradiction and $a+g=1$.
(5). For $a \geq 1$, lemma 3.3 tells us that Q must have been open with $mci_Q=y$, hence we have a contradiction and $a=0$. Hence $g=1$ and $y=0$. ■

4. Loss and duplication.

In order to derive formal results on the loss and duplication problems in the protocol skeleton, we need a formal definition of these problems in terms of the parameters of the protocol skeleton. Informally we talk about "loss" if P sends Q a message and thinks it arrived, while Q has not received it. We talk about "duplication" if Q receives a message from P and treats it as a new message while it really was a retransmission of an old message.

4.1. Correct communication.

Definition 4.1. A successful (m,y,g) close for P is an (m,y,g,a) close for P with $a \geq cs_{\mathrm{P}}$. The predicate $sclosed_{\mathrm{P}}(m,y,g)$ becomes true when P does a successful (m,y,g) close.
Loss is the situation in which P does a successful $(m,y,1)$ close while Q never does a successful (m',y',g) close with $m=y'$.
Duplication is the situation in which P does a successful $(m,y,0)$ close while Q never does a successful (m',y',g) close with $m=y'$.
Correct communication is the situation in which P does a successful (m,y,g) close iff Q does a successful (y,m,g') close.

The definition of loss is clearly reasonable. Consider the situation in which duplication can arise. Typically Q sends an opening packet $<mci_{\mathrm{Q}}(t_0),0,0>$ twice. P, upon receipt of the first one at t_1, opens with $g_{\mathrm{P}}(t_1)=0$, $mci_{\mathrm{P}}(t_1):=new\ value$ and $yci_{\mathrm{P}}(t_1)=mci_{\mathrm{Q}}(t_0)$. If P closes successfully (possibly after more packets exchanges), and after that, at time t_2 receives Q's retransmission of the original opening packet, it will open with $g_{\mathrm{P}}(t_2)=0$, $mci_{\mathrm{P}}(t_2):=new\ value$, and hence $mci_{\mathrm{P}}(t_2)\neq mci_{\mathrm{P}}(t_1)$, and $yci_{\mathrm{P}}(t_2)=mci_{\mathrm{Q}}(t_0)$. If P closes successfully again, we have the duplication problem. Now consider Q. Q can only do a successful (m,y,g) close with $m=mci_{\mathrm{Q}}(t_0)$ once since, if it closes and opens again, $mci_{\mathrm{Q}}:=new\ value$. Thus either P's successful $(mci_{\mathrm{P}}(t_1),yci_{\mathrm{P}}(t_1),0)$ close does not correspond to a successful $(mci_{\mathrm{Q}}(t_0),mci_{\mathrm{P}}(t_1),g)$ close or P's successful $(mci_{\mathrm{P}}(t_2),yci_{\mathrm{P}}(t_2),0)$ close does not correspond to a successful $(mci_{\mathrm{Q}}(t_0),mci_{\mathrm{P}}(t_2),g)$ close.

Note that correct communication does not mean that the relation

$$sclosed_{\mathrm{P}}(m,y,g) \Leftrightarrow sclosed_{\mathrm{Q}}(y,m,g')$$

is invariant, e.g. P may close first while Q is still open; thus the relation does not hold without referring to (possibly different) time moments. However, correct communication means there is some time that the relation will hold.

Theorem 4.1.
(1) Correct communication implies no loss and no duplication.
(2) If no loss and no duplication occurs during a finite number of successful closes, then we have correct communication.
(3) Correct communication preserves order.

Proof. (1). Obvious.
(2). Let P do a successful (m_0,y_0,g_0) close C_0. No loss or duplication implies Q does a successful (m_1,y_1,g_1) close C_1 with $y_1=m_0$. Assume $y_0\neq m_1$. Then (lemma 3.7 (2)) $g_1=0$. In general, for $i\geq 1$, let one processor do a successful $(m_{i-1},y_{i-1},g_{i-1})$ close C_{i-1}, while the other processor does a successful (m_i,y_i,g_i) close C_i with $m_i\neq y_{i-1}$. Then $g_i=0$, and because of no loss and no duplication, the first processor does a successful $(m_{i+1},y_{i+1},g_{i+1})$ close C_{i+1} with $y_{i+1}=m_i$. Since $y_{i+1}\neq y_{i-1}$, we have $m_{i+1}\neq m_{i-1}$ and $g_{i+1}=0$. Let t_i be the time that the connection with $mci=m_i$ is opened. Since $g_i=0$, $a_i(t_i)=1$ and there exists a time t, $t<t_i$, that the other processor is open with $mci=y_i$. Because $y_i=m_{i-1}$, $t<t_i$ and $t_{i-1}\leq t$, we have $t_{i-1}<t_i$. Since also $t_i<t_{i+1}$, $C_{i-1}\neq C_{i+1}$. Consider the sequence of closes C_0, C_1, ... defined by the condition that there are no losses and duplication. Then we have that for each C_i, $i\geq 1$, there is a C_{i+1} with $y_{i+1}=m_i$, $g_{i+1}=0$, $t_{i+1}>t_j$ for all j, $0\leq j\leq i$ and thus $C_{i+1}\neq C_j$ for all j, $0\leq j\leq i$. Hence

this sequence is infinite. Contradiction. Thus $y_0 = m_1$ and P's successful (m_0, y_0, g_0) close implies a successful (y_0, m_0, g_1) close by Q. The same argument holds for the reverse implication.
(3). Let P do a successful (m_0, y_0, g_0) close at time t_0, and a successful (m_1, y_1, g_1) close at time t_1. Correct communication implies that Q does a successful (y_0, m_0, g) close, and a successful (y_1, m_1, g') close, say at times t_2 and t_3 respectively. Assume without loss of generality that t_0 is the smallest value. From lemmas 3.3 and 3.6 follows that between t_0 and t_2 $closed_P(yci_Q, mci_Q, g, a)$ and $\neg closed_Q(m'', mci_P, g'', a'')$ holds. Thus also $\neg closed_Q(y_1, m_1, g', a')$ holds. So $t_3 > t_2$, and as $t_1 > t_0$, the order is preserved. ■

We note that the condition of a finite number of closes in theorem 4.1 (2) is only necessary in the general case where no assumption is made yet on the value of the parameter cs. It is clear from the proof that this condition can be dropped if it is known that $cs > 1 - g$.

4.2. Parameters for closing.

Lemma 4.2. To avoid the loss and duplication problems, without further assumptions on the order of operations, it is required that
(1) $cf_P \geq 1 - g_P$,
(2) $cs_P > cf_Q + 1 - g_P$,
(3) $cs_P \geq 2$.

Proof. (1). It is clear from lemma 3.7 that if P wants to close while $a_P + g_P = 1$, P had better do it as a failure, since Q might not have opened the corresponding connection at all. (If P would not be able to close, deadlock would arise.) Hence $cf_P + g_P \geq 1$ and $cf_P \geq 1 - g_P$.
(2). If P does an (m, y, g, cs_P) close, and Q does an (m', y', g', a') close with $m = y'$ and $y = m'$, we know with lemma 3.7 that it might be the case that $a' = cs_P + g_P - 1$. Hence we need $cf_Q < a' = cs_P + g_P - 1$ to avoid the loss and duplication problems, and thus $cs_P > cf_Q + 1 - g_P$.
(3). Substituting the minimal value for cf_Q gives $cs_P > cf_Q + 1 - g_P \geq 1 - g_Q + 1 - g_P \geq 1$ since $g_P + g_Q \leq 1$. Hence $cs_P \geq 2$. ■

Consider the minimal values we can choose for cf_P and cs_P: $cf_P = 1 - g_P$ and $cs_P = 2$. Now if $g_P = 0$, $cs_P = cf_P + 1$, and for each value of a_P P knows how to close, as a failure or a success. But if $g_P = 1$, $cf_P = 0$ and $cs_P = 2$. What should P report: success or failure, if it closes with $a_P = 1$? If P reports success we have the possibility of loss. If P reports failure, we have the possibility of duplication. So P had better not close at all if $a_P = g_P = 1$. But this leaves us with the following problem: assume $a_P = g_P = 1$ and P sends a packet with $seq = 1$ to Q. Upon receipt of this, Q puts a_Q to 2 and closes successfully. Now P will never get a packet with right mci and yci fields and $seq = 1$ anymore, hence a_P cannot rise to 2. The first idea which comes to mind to repair this deadlock problem is to forbid Q to close with $a_Q = 2$ and wait with closing until $a_Q = 3$. Now P is o.k. but Q might have a problem if it does not receive the valid packet from P to enable it to set a_Q to 3. But there is one difference: if Q is forced to close with $a_Q = 2$, it does know how, namely successfully. Q can use this in the following way: it keeps on retransmitting its packet with $seq = 1$ (this cannot cause confusion later because it contains a $yci_Q = mci_P$). Now if P is still open (and the link is still up) P will eventually respond with a valid packet with $seq=2$, so Q can set a_Q to 3 and close. Note that this packet exchange is a 5-way handshake. On the other hand, if P had already closed, it will send an error packet to Q that it is not open any more with the value $mci_P = yci_Q$ (even if P opened again in the mean time). But if Q receives this error packet, it knows that a_P has been 2 and that P closed successfully because P would not have closed when $a_P = 1$. Hence Q can safely close successfully.

Now this works if processors stay up, and if we assume that some packet will eventually get through if we keep on trying. That it does not work if processors go down should not bother us too much, since it is not possible to design a protocol that always works correctly in the presence of processor breakdowns anyway (cf. [1]). The reason we need a five-way handshake is that P cannot use the same trick as Q does: waiting for an error packet. If P received an error packet to the effect that Q had already closed while $a_P = 1$, it cannot decide whether Q closed as a failure (with $a_Q = 1$) or as a success (with $a_Q = 2$). Note that the extra information that is used by the processors, except the information deriving from the protocol skeleton, is information about when a connection is closed and when it is not. But if we want to use this kind of information, there is no need to restrict it to 'P does not close with $a_P = 1$ if $g_P = 1$'. It is more fruitful to demand that no processor closes 'arbitrarily'. In order to be able to define the notion of a nonarbitrary close precisely, we need an analysis of possible error packets and their consequences.

5. Error handling.

In order to analyze the effect of sending error packets, we will begin with assuming that error packets are sent whenever the procedure error is called, and that the actual error packet contains all information the other party might need. Since we will see later that not all information is used, nor that all error packets sent are meaningful, we can then decide what fields the error packet should contain and in which circumstances an error packet should be sent.

We remark that the informal proof given above that the five-way handshake works in the absence of processor failures, if P and Q know they will not do arbitrary closes, is not watertight and in fact contains a flaw, as we will see in the sequel. Unfortunately, Belsnes [1] did make this mistake in the analysis of his four-way handshake. The problem is the following: Q, being in a state with $a_Q = 2$ and receiving an error packet from P stating that P is not open with mci_P, concludes that P has closed. But it might be the case that this error packet was sent before the packet exchange which led to $a_Q = 2$ took place. In that case, P might be open still, even with $a_P = 1$, and Q's conclusion that P closed successfully is not warranted. For the time being, we define the procedure error as follows.

procedure $error(<x, y, z>)$ = send $<\text{error}, mci/y, yci/x, a/z>$

It is understood here that if the processor which sends the error packet is closed, it fills in 'undefined' for mci, yci and a. Note that lemmas 3.1 up till 4.2 (1) hold irrespective of the closing strategy and error procedure used, but that lemmas 4.2 (2) and 4.2 (3) might not hold any more, since restricting the protocol skeleton gives the processors extra information.

5.1. Invalid packets.

Lemma 5.1. Let P receive a packet $<x,y,z>$ from Q which is not valid. Then
(1) $y \neq 0 \Rightarrow (mci_P = y \vee closed_P(y, y', g', a'))$.
(2) $z \geq 1 \Rightarrow ((mci_P = y \wedge yci_P = x) \vee closed_P(y, x, g', a'))$.
(3) $mci_P = \text{undefined} \Rightarrow z \geq 1$.
(4) $(mci_P \neq \text{undefined} \wedge valid_P(y)) \Rightarrow$
$$(y = mci_P \wedge z < a_P \wedge ((valid_P(x) \wedge accepted_P(<x, y', z>)) \vee z = 0)).$$
(5) $(mci_P \neq \text{undefined} \wedge valid_P(x) \wedge y = g_P = 0) \Rightarrow accepted_P(<x,y,z>)$.

Proof. By assumption $sent_Q(<x,y,z>)$ and $\neg valid_P(<x,y,z>)$ hold.
(1). Use lemma 3.2 and 3.3.
(2). Use lemma 3.2 and 3.5.
(3). If P is closed, every packet with $z=0$ is accepted as valid, hence $z \geq 1$.
(4). Thus $y = mci_P$. There are two cases.
Case 1. $z=0$. Let $a_P=0$. Then x is valid and z is valid which leads to a contradiction. Thus $a_P \geq 1$ and $a_P > z$.
Case 2. $z \geq 1$. With lemma 5.1 (2) we have that $y = mci_P$ and $x = yci_P$. Thus x is valid and necessarily, z is not. Using lemma 3.2 and 3.5 we conclude that $z \leq a_P$. Hence $z < a_P$.
As we know from (4), $z < a_P(t)$, hence $a_P(t) \geq 1$. Thus x valid implies $x = yci_P$. Since $a_P(t)$ increases one by one and $z < a_P(t)$, we have $accepted_P(<x, y', z>)$ with lemma 3.3.
(5). Since $g_P = 0$ we have $a_P \geq 1$, and $valid_P(x)$ implies $x = yci_P$. As $y=0$, $z=0$, too. Use lemma 3.3. ■

Note that in the last case of (4), where $mci_P \neq$undefined, $valid_P(y)$, $\neg valid_P(x)$ and thus $z=0$, it might be the case that Q opened twice with $yci_Q = mci_P = y$. Since $a_P > z$, $a_P \geq 1$ and $mci_Q = yci_P$ or $closed_Q(yci_P, y', g', a')$. $sent_Q(<x,y,z>)$ implies $(mci_Q = x \wedge yci_Q = y)$ or $closed_Q(x, y, g', a')$. If $y' = y$, Q has opened twice with $yci_Q = y$, since $x \neq yci_P$.

5.2. Error packets. Because of lemma 5.1 (5), we now require that error packets are only sent upon receipt of an invalid packet in case a copy of this packet was not accepted before during this connection. In the case it was, P had better retransmit its last packet, because that probably got lost.

Since P does not know, nor can do anything about previous connections, it is clearly reasonable that P only considers error packets pertaining to its current connection.

Lemma 5.2. Let P receive an error packet $<\text{error}, m'/y, y'/x, a'/z>$ from Q with $x = mci_P$. Then
(1) $m' = y \Rightarrow (z = g_P = 0 \wedge y = yci_P \wedge a_P = 1 \wedge \neg(mci_Q = yci_P \wedge yci_Q = mci_P) \wedge$
$\neg closed_Q(yci_P, mci_P, g, a))$.
(2) $(m' \neq y \wedge y \neq 0) \Rightarrow (y = yci_P \wedge closed_Q(y, y'', g, a))$.
(3) $(y = 0 \wedge yci_P \neq 0) \Rightarrow (m' \neq yci_P \wedge m' \neq \text{undefined} \wedge$
$((mci_Q = yci_P \wedge closed_Q(m', y'', g, a)) \vee$
$(mci_Q = m' \wedge closed_Q(yci_P, y''', g'', a'')) \vee$
$(closed_Q(m', y'', g, a) \wedge closed_Q(yci_P, y''', g'', a''')))$.

Proof. (1). Since $m'=y$, we have $valid_Q(y)$, $y\neq 0$ and $m'\neq$undefined. Since $x=mci_P$ and $Y\neq 0$, $y=yci_P=m'$. Since error packets are not sent if a copy of the invalid packet was already accepted during this connection, we know with lemma 5.1 (4) that $\neg valid_Q(x)\wedge z=0$. Moreover, at that time $a_Q\geq 1$. Thus $(mci_Q=m' \wedge yci_Q\neq 0\wedge yci_Q\neq x) \vee (closed_Q(m',y',g,a)\wedge y'\neq x)$. The assumption $a_P+g_P\geq 2$ leads to a contradiction. Thus $a_P+g_P=1$, and since $y=yci_P=mci_Q$, $a_P=1$ and $g_P=0$.
(2). Since $x=mci_P$ and $y\neq 0$, $yci_P=y$ and $a_P\geq 1$. At the time $<x,y,z>$ was sent, $y\neq 0$ already, thus $mci_Q=yci_P\vee closed_Q(yci_P,y'',g,a)$. Since $m'\neq y$ at the time $<x,y,z>$ was received by Q, $closed_Q(yci_P,y'',g,a)$ already was true. Hence it still is.
(3). Because $y=0$, we have $g_P=1$, $m'\neq y$ and $m'\neq$undefined. Since Q sent an error packet, at that time $\neg(g_Q=0\wedge x=yci_Q)$. Assume $g_Q=1$. Then $m'\neq yci_P$ (otherwise $g_P+g_Q\leq 1$). Assume $x\neq yci_Q$. Assume also $m'=yci_P$. Then $a_P\geq 1$ and $a_P+g_P\geq 2$ and lemma 3.5 leads to a contradiction. Thus $m'\neq yci_P$ in both cases. Hence relation (3) holds. ■

Thus, in case (3) of lemma 5.2, P cannot draw the conclusion that Q already has closed the connection with $mci_Q=yci_P$.

Definition 5.1. An error (m,y,g,a) close for P is an (m,y,g,a) close for P upon receipt of an error packet <error,$m'/y,y'/m,a'/z$>, i.e. without doing any other operations in between. The predicate $eclosed_P(m,y,g,a)$ becomes true when P does an error (m,y,g,a) close.

The reason that we not only require $mci_P=m$ but also $yci_P=y$ for a close, is that in the case where $a+g\geq 2$ (and P knows that Q has been open with $mci_Q=yci_P$ and $mci_P=yci_Q$), P would like to conclude 'Q has already closed this connection, so I had better close too'. We know from lemma 5.2 that this conclusion is only warranted if $y=yci_P$.

Lemma 5.3. $a_P+g_P\geq 2 \Rightarrow \neg eclosed_Q(yci_P,y',g',a')$.

Proof. Assume $a_P+g_P\geq 2$ and $eclosed_Q(yci_P,y',g',a')$. Thus $closed_Q(yci_P,y',g',a')$ and with lemma 3.5 we have $y'=mci_P$ and $a'\geq 1$. Thus Q received the error packet <error,$m''/mci_P,y''/yci_P,a''/z$>. Thus $m''\neq mci_P$ otherwise P would not have sent the error packet. With lemma 5.2 (2) we have $closed_P(mci_P,y'',g'',a'')$. Since P is open still, we have a contradiction. ■

5.3. Arbitrary closes. In order to be able to define an arbitrary close we introduce a new variable *last* which contains the number of different packets which should be received by one processor during one connection.

Definition 5.2. Let P do an (m,y,g,a) close. The close is called *arbitrary* if none of the following hold:
(1) P goes down (*break-down close*),
(2) $eclosed_P(m,y,g,a)$,
(3) $a\geq last_P$.

Case (3) corresponds to a 'complete' packet exchange in this connection. For example, a 4-way handshake would correspond to $last=2$ for both sender and receiver. The parameter $last_P$ possibly depends on g_P, or else it is a constant.

It is clear from the above that most information included in error packets is not used at all. Consider an error packet that P might receive: <error,$mci_Q/y, yci_Q/x, a_Q/z$>. P needs x and y to test against mci_P and yci_P to avoid a nonarbitrary close. a_Q and z were never used throughout the analysis, hence we can discard them. Although the values of mci_Q and yci_Q give some information about the state in which Q was, the decision of P which action to take should not depend on that information, but its value of a_P and g_P. The following cases arise.
Case 1. $a_P+g_P \geq 2$. Then Q has closed the corresponding connection and it is irrelevant for P's current connection whether Q has opened again. P should close successfully.
Case 2. $a_P=1$, $g_P=0$. P should close and report failure to avoid duplication.
Case 3. $a_P=0$, $g_P=1$. Then Q is open with another connection and is not ready to reply to this new one. P could wait and try again or close as a failure.

Hence an error packet of the form <error,y,x> is sufficient, always assuming error packets are only sent upon invalid packets which were not already accepted during the same connection. Reconsidering the fields in normal packets $<x,y,z>$ and lemma 5.1 (4)-(5), one might be tempted to include the field x only in packets with $z=0$, since the receiver knows that it must be valid if y is valid and $z \geq 1$. But that does not work because, in case y happened to be invalid, the error packet must contain x. Thus we redefine the procedure *error* as follows.

```
procedure error(<x,y,z>) =
    if connection with Q closed then send <error,y,x>
    else  if (($x=yci_P$ or $a_P=0$) and ($y=mci_P$)) then skip
          else send <error,y,x>
```

Summarizing, we can state the implications of closing upon receipt of an error packet.

Lemma 5.4.
(1) $(eclosed_P(m,y,g,a) \wedge a+g \geq 2) \Rightarrow$
$(closed_Q(y,m,g',a') \wedge a'+g' \geq a \wedge \neg\, eclosed_Q(y,m,g',a'))$.
(2) $eclosed_P(m,y,0,1) \Rightarrow (closed_Q(y,m,g',a') \Rightarrow a' \leq 1)$.
(3) $eclosed_P(m,0,1,0) \Rightarrow (closed_Q(m',m,g',a') \Rightarrow a' \leq 1)$.

Proof. (1). Apply lemmas 3.5, 5.2 and 5.3. (2). Apply lemmas 5.2 and 3.7. (3). Apply lemma 3.7. ■

6. The selection of parameters.

6.1. The four-way handshake. The crucial theorem for k-way handshakes for connection management can now be formulated as follows:

Theorem 6.1. Let the following four conditions hold:
(1) connections do not stay open indefinitely,

(2) there are no processor breakdowns,
(3) there are no arbitrary closes,
(4) $last_P \geq 2$, $last_Q \geq 2$.

Then P does an (m,y,g,a) close with $a+g \geq 2$ iff
Q does a (y,m,g',a') close with $a'+g' \geq 2$.

Proof. Note that the conditions (3) and (4) ensure that the following relation is invariant:

$$closed_P(m,y,g,a) \Rightarrow (eclosed_P(m,y,g,a)) \vee a \geq last_P).$$

We first show the "only if" part of the theorem. Suppose P does an (m,y,g,a) close with $a+g \geq 2$. We have two cases.

Case 1. P closed because $a_P \geq last_P$. Hence $a \geq 2$ and with lemma 3.5, $mci_Q = y \wedge yci_Q = m \wedge 2 \leq a_Q + g_Q$ or $closed_Q(y,m,g',a') \wedge 2 \leq a'+g'$. Since Q will not stay open indefinitely, Q will eventually do a (y,m,g',a') close with $a'+g' \geq 2$.

Case 2. $eclosed_P(m,y,g,a)$ holds. Hence (lemma 5.4), $closed_Q(y,m,g',a') \wedge \neg\, eclosed_Q(y,m,g',a')$. Thus $a' \geq last_Q$. Hence $a' \geq 2$, and thus $a'+g' \geq 2$.

Next suppose Q does a (y,m,g',a') close with $a'+g' \geq 2$. Then we can use the same argument with P and Q interchanged. ■

Thus the Tomlinson handshake [7] does not ensure correct communication, since it is a special instance of the 3-way handshake from lemma 6.3.

Corollary 6.2. Under the assumptions from theorem 6.1, we can achieve correct communication by taking $cs = 2-g$.

Proof. Take $cf_P = 1-g_P$, $cf_Q = 1-g_Q$, $cs_P = 2-g_P$ and $cs_Q = 2-g_Q$. Then every (m,y,g,a) close with $a+g \geq 2$ is successful and vice versa. Thus theorem 6.1 ensures correct communication. ■

Note that we did not exclude link failures in the requirements for theorem 6.1. Link failures are handled by just leaving connections open until links come up again. If a processor cannot see whether a link is temporarily disabled, there is no difference with the case that all packets sent in a certain time interval are lost.

Lemma 6.3. Let the following three conditions hold:
(1) connections do not stay open indefinitely,
(2) there are no processor breakdowns,
(3) there are no arbitrary closes.

Then it is not possible to avoid loss and duplication problems by taking $last = 2-g$, without further assumptions.

Proof. Let P open with $a_P = 0$ and $g_P = 1$ and send an opening packet $<mci_P,0,0>$ to Q. Q opens with $a_Q = 1$ and $g_Q = 0$, $mci_Q := new\ value$ and $yci_Q := mci_P$. Q replies with $<mci_Q, yci_Q, 0>$ and retransmits after some time. Now Q receives an error packet $<$error$, mci_Q, yci_Q>$ because P has closed in the mean time. What should Q do? Q knows that P has been open with $mci_P = yci_Q$ and P has not closed arbitrarily. There are three things which could have happened, and unfortunately Q has no way to decide which did.

Case 1. P received Q's packet and closed with $a_P+g_P=2$, but P's reply packet got lost. Since P closed successfully, Q should close as a success too (otherwise we have a loss).
Case 2. P has closed with $a_P+g_P=2$, but on a different packet from Q, since this was the second connection for Q with $yci_Q=mci_P$. Hence Q should close as a failure, otherwise we have a duplication.
Case 3. P has closed with $a_P+g_P=1$, on receipt of an error packet from Q, sent during an earlier connection, e.g. when P and Q tried to start up a connection simultaneously. Since P closed as a failure, Q should too, otherwise we have a duplication.
Thus there is no way to choose the parameters cf and cs such that correct communication is achieved in all three cases. ■

Corollary 6.4. For correct communication using k-way handshakes in the absence of processor breakdowns, it is necessary as well as sufficient that $k \geq 4$.

6.2. Discussion.

The obvious way to choose *last* is either $last=k$, leading to a $(2k)$-way handshake (an 'even handshake'), or $last=k-g$, leading to a $(2k-1)$-way handshake (an 'odd handshake'). However, the protocol skeleton would work also with, for example, $last=k-2g$. This choice would even lead to correct communication under the conditions of theorem 5.4. It does however have the drawback that a successful packet exchange now relies more on error packets which might have to be sent even if no packets get lost. Let $k=4$, $g_P=1$ and $g_Q=0$. Then P can close when $a_P=2$, but Q should only close with $a_Q=4$. However, Q will never set a_Q to 4, since it needs a packet for that which P is only allowed to send when $a_P=3$. Hence Q always needs an error packet for closing, unless P does not close with $a_P=2$. (Although P is allowed to close in that state, the protocol skeleton does not force P to close.)

There is a difference between even and odd handshakes, as Belsnes [1] already pointed out, which might be important in practical cases. Both work correctly in the absence of processor breakdowns, if $last \geq 2$. If the last packet is lost, it is substituted as it were, with an error packet. Now the receiver of the error packet concludes that the other processor has closed successfully and closes successfully too. If however, the last packet was not lost, but the processor which had to send it went down, we either have a loss or a duplication problem. For an even handshake, the last packet sent goes from the processor with $g=0$ to the one with $g=1$, hence we might have loss of packets. In the case of an odd handshake, the last packet goes from the processor with $g=1$ to the one with $g=0$, hence we might get duplication. Hence if, in a practical situation, a loss is more disastrous than a duplication, one might consider whether the loss in efficiency caused by using a 5-way instead of a 4-way handshake is outweighed by the advantage of avoiding loss instead of duplication.

Up till now, we never considered in which packet during the packet exchange the actual message which had to be communicated, was incorporated. As we can see from the analysis, it really does not matter, as long as we do not use the very last packet in case of an odd handshake. It can only be argued that this might tempt a processor to unfair play.

As we restricted ourselves to a protocol skeleton, it is clear that for an actual implementation there remains a lot to be specified before getting a working protocol. For example, a time-out mechanism should be added to control the retransmissions of packets, and some order of operations should be defined. Note that there is some freedom in the specification of the protocol skeleton that does not contribute to an efficient correct communication. For example, the possibility to send packets with a sequence number strictly smaller than $a+g-1$ does not contribute to the communication. Nor does it help if closing is postponed when $a \geq last$. A restriction of the protocol skeleton which would reduce the number of erroneous openings is the following. If a closed processor receives a packet with $z=0$, it now always opens because it considers it as a opening packet. However, it also could be a reply to an opening packet. The latter case can be excluded by testing the y-field: real opening packets contain a y-field equal 0. The reason we did not incorporate all these restrictions in the protocol skeleton is, that it is not necessary for the proof and leaves the basic structure visible which is responsible for the desired property of the protocol, namely reliable connection management.

The advantage of such a general set-up is primarily that the proofs capture all protocols which can be viewed as instances of the protocol skeleton. The next section contains examples of extensions of this basic protocol skeleton for multiple message exchanges. Thus we know how far we can get towards a reliable connection management by a certain setting of parameters. Secondly, if we want more, e.g. reliable communication in the presence of processor breakdowns, we know that we should either devise a protocol skeleton based on a different principle, or else relax the assumptions. In section 8 we will investigate the effect of relaxing several assumptions. Thirdly, we have learned from this analysis that the problem with connection management under these stated assumptions, namely that any last packet in a finite packet exchange can be lost, is solved partly by allowing that certain error packets are substituted for the last packet.

Hence we strongly advocate a modular protocol design, to make separate proofs of the correctness of different aspects of protocol performance possible. As an illustration, we extend the basic protocol skeleton for use for one and two-sided multiple-message communication.

7. Extensions for multiple-message communication.

If we state theorem 6.1 in a slightly more general version, it is easily seen that this protocol skeleton not only handles single-message communication correctly, but also multiple-message communication. Let n, $n \geq 1$, be the number of messages to be transmitted.

Lemma 7.1. Let the following four conditions hold:

(1) connections do not stay open indefinitely,

(2) there are no processor breakdowns,

(3) there are no arbitrary closes,

(4) $last_P \geq n+1$, $last_Q \geq n+1$.

Then P does an (m,y,g,a) close with $a+g \geq n+1$ iff
Q does an (y,m,g',a') close with $a'+g' \geq n+1$.

Proof. Proof of theorem 6.1 with 2 replaced by $n+1$. ■

7.1. One-sided multiple-message communication.

It is usually not the case that n is a constant, and we would like to be able to choose n different for each connection. It is possible to incorporate this feature in the protocol skeleton, by including the value of n in the opening packet of the sender. Thus we formulate an extended protocol skeleton for multiple-message communication, consisting of the three basic operations S^1, R^1 and C^1. We include the messages $D[1]$, ..., $D[n]$ to be sent and the nonarbitrary close. Note that, in contrast to e.g. a sliding-window protocol, we need that the message field and the *seq* field in a packet are large enough to contain the value $n+1$, and that we cannot use sequence numbers modulo some value. This is due to the assumption that arbitrary delays are possible, while the sliding-window protocol was designed for links which have the FIFO property.

```
$S_P^1$:  if connection with Q closed then        (*open*)
          begin $mci_P$:=$new\ value$; $yci_P$:=0; $g_P$:=1; $a_P$:=0; $last_P$:=$n$+1; $D[0]$:=$n$ end;
          if $g=1$ then send a packet <$mci_P, yci_P, seq, D[seq]$> to Q
                  else send a packet <$mci_P, yci_P, seq, \varnothing$> to Q
          where $0 \le seq < a_P + g_P$.

$R_P^1$:  receive <$x,y,z,d$>;
          if connection with Q closed
          then if $y \ne 0$ then $error$ (<$x,y,z,d$>)
               else begin $mci_P$:=$new\ value$; $yci_P$:=$x$; $g_P$:=0; $a_P$:=1; $last_P$:=$d$+1 end
          else if not (($x=yci_P$ or $a_P=0$)        (*x valid*)
                   and $y=mci_P$                     (*y valid*)
                   and $z=a_P$)                      (*z valid*)
               then $error$ (<$x,y,z,d$>)
               else begin $a_P$:=$a_P$+1; if $z$=0 then $yci_P$:=$x$ end

$C_P^1$:  if connection with Q is open then
          if <error,$yci_P, mci_P$> received or $a_P \ge last_P$ then
          begin if $a_P \le cf_P$ then report failure;
               if $a_P \ge cs_P$ then report success;
               $mci_P$:=undefined; $yci_P$:=undefined; $a_P$:=undefined;
               $g_P$:=undefined; $last_P$:=undefined
          end
```

where $cf_P = 1 - g_P$ and $cs_P = 2 - g_P$.

We remark that all lemmas except lemmas 4.2 (2) and 4.2 (3) still hold for this multiple-message protocol skeleton. In addition we need the following notation and lemma about *last*, which now has become a variable.

Definition 7.1. An (m,y,g,a,l) *close* for P is an operation C_P^1, invoked at some time t by P, such that $mci_P(t)=m$, $yci_P(t)=y$, $g_P(t)=g$, $a_P(t)=a$ and $last_P(t)=l$ (We consider only meaningful closes, i.e. $m \ne$undefined.) The predicate $closed_P(m,y,g,a,l)$ becomes true when P does an (m,y,g,a,l) close.

Lemma 7.2. Let P and Q operate with the multiple-message protocol skeleton. Then

(1) If P has an open connection with Q at time t and t_1 with $mci_P(t)=mci_P(t_1)$, then $last_P(t)=last_P(t_1)$,

(2) $a_P \ge 1 \Rightarrow ((mci_Q = yci_P \wedge last_P = last_Q) \vee (closed_Q(yci_P, y', g', a', last_P))$.

Proof. Obvious from the protocol skeleton. ■

Theorem 7.3. Let P and Q operate with the multiple-message protocol skeleton. Let the following two conditions hold:
(1) connections do not stay open indefinitely,
(2) there are no processor breakdowns,
Then P does an $(m, y, g, a, n+1)$ close with $a+g \geq 2$ implies
Q does an $(y, m, g', a', n+1)$ close with $a'+g' \geq n+1 \geq 2$.

Proof. Use lemma 7.1 and 7.2. ■

Corollary 7.4. Under the assumptions of theorem 7.3 we can correctly communicate finite, nonempty message sequences with the protocol skeleton consisting of operations S^1, R^1 and C^1.

Proof. The only thing left to check is that inside one connection no loss or duplication of messages occurs. It is clear from operation R^1 that message i is accepted when a is set to $i+1$. Since a is increased one by one, all messages $D[1]$, ..., $D[k]$ are accepted exactly once if the processor closes with $a=k+1$. Since for the receiver $g=0$, $a+g \geq n+1$ implies that all n messages are accepted. ■

Note that, although we need a 4-way handshake to send a single message, we do not need a $4n$-way handshake to send n messages, but only a $2(n+1)$-way handshake.

7.2. Two-sided multiple-message communication. Another extension we can immediately make is to allow the message transfer in a connection to be two sided. In order to decide how long a packet exchange needs to be in the case that the 'receiver' has more messages to send than the 'sender', we need the corresponding version of lemma 7.1 for odd handshakes in the basic protocol skeleton (lemma 7.1 corresponds to even handshakes).

Lemma 7.5. Let the following four conditions hold:
(1) connections do not stay open indefinitely,
(2) there are no processor breakdowns,
(3) there are no arbitrary closes,
(4) $last_P \geq n+2-g_P$, $last_Q \geq n+2-g_Q$.
Then P does an (m, y, g, a) close with $a \geq n+1$ iff
Q does an (y, m, g', a') close with $a' \geq n+1$.

Proof. Analogous to the proof of lemma 7.1. ■

Since an odd handshake is most efficient when the 'sender' has less to send than the 'receiver', and an even handshake is most efficient when the 'sender' has more to send, we can let the protocol skeleton decide on the spot. Thus we extend the protocol skeleton as follows.

```
S_P^2: if connection with Q closed then      (*open*)
       begin $mci_P$:=new value; $yci_P$:=0; $g_P$:=1; $a_P$:=0; $last_P$:= $n_P$+1; $D[0]$:= $n_P$ end;
       send a packet <$mci_P$, $yci_P$, $seq$, $D[seq]$> to Q where $0 \leq seq < a_P+g_P$.
```

```
$R_P^2$: receive $<x,y,z,d>$;
     if connection with Q closed
     then if $z \neq 0$ then $error$ $(<x,y,z,d>)$
          else begin $mci_P := new\ value$; $yci_P := x$; $g_P := 0$; $a_P := 1$;
                     $D[0] := n_P$; $last_P := max(d+1, n_P+2)$
               end
     else if not $((x = yci_P$ or $a_P = 0)$      (*x valid*)
                 and $y = mci_P$                  (*y valid*)
                 and $z = a_P)$                   (*z valid*)
          then $error$ $(<x,y,z,d>)$
          else begin $a_P := a_P + 1$;
                     if $z = 0$ then begin $yci_P := x$; $last_P := max(d+1, n_P+2-g_P)$ end
               end

$C_P^2$: if connection with Q is open then
     if <error, $yci_P, mci_P$> received or $a_P \geq last_P$ then
     begin if $a_P \leq cf_P$ then report failure;
           if $a_P \geq cs_P$ then report success;
           $mci_P$:=undefined; $yci_P$:=undefined; $a_P$:=undefined;
           $g_P$:=undefined; $last_P$:=undefined
     end
```

where n_P is the number of messages to be sent over by P. Note that, instead of $n \geq 1$ for the one-way case, we now have $n_P \geq g_P$, hence the 'sender' transmits a nonempty sequence, while the 'receiver' may transmit an empty sequence of messages. Again, $cf_P = 1 - g_P$ and $cs_P = 2 - g_P$. We assume $D[seq] = \emptyset$ for $seq > n_P$. We have the following supplementing notation and lemma about *last*.

Definition 7.2. An (m,y,g,a,l,n) *close* for P is an operation C_P^2, invoked at some time t by P, such that $mci_P(t) = m$, $yci_P(t) = y$, $g_P(t) = g$, $a_P(t) = a$, $last_P(t) = l$ and $n_P(t) = n$ (We consider only meaningful closes, i.e. $m \neq$undefined.) The predicate $closed_P(m,y,g,a,l,n)$ becomes true when P does an (m,y,g,a,l,n) close.

Lemma 7.6. Let P and Q operate with the two-sided multiple-message protocol skeleton. Then

(1) If P has an open connection with Q at time t_1 and t, $t_1 < t$, with $mci_P(t) = mci_P(t_1)$ and $a_P(t_1) \geq 1$, then $last_P(t) = last_P(t_1)$,

(2) $mci_P \neq$undefined $\Rightarrow last_P \geq n_P + 2 - g_P$,

(3) $a_P + g_P = 1 \Rightarrow last_P > a_P$,

(4) $a_P + g_P \geq 2 \Rightarrow$

$$((mci_Q = yci_P \wedge yci_Q = mci_P \wedge last_P = last_Q) \vee closed_Q(yci_P, mci_P, g', a', last_P, n')).$$

Proof. Obvious. ■

Theorem 7.7. Let P and Q operate with the two-sided multiple-message protocol skeleton. Let the following two conditions hold:

(1) connections do not stay open indefinitely,

(2) there are no processor breakdowns,

Then P does an (m,y,g,a,l,n) close with $a+g \geq 2$ implies
Q does an (y,m,g',a',l,n') close at some time with $a' \geq n+1$ and $a \geq n'+1$.

Proof. Use lemmas 7.5 and 7.6. ∎

Corollary 7.8. Under the assumptions of theorem 7.7 we can correctly communicate finite, nonempty message sequences in both directions at once with the protocol skeleton consisting of operations S^2, R^2 and C^2.

8. Relaxing assumptions.

A 4-way handshake might be too high a price for correct communication, especially since correctness is only guaranteed if processors do not go down while they have an open connection. Therefore we investigate the question how the assumptions can be relaxed while keeping the communication as reliable as possible. There are basicly four ways to relax the assumptions.

Firstly, we could allow the processors to remember information about previous connections. This increases the memory requirements of the processors, and makes communication more sensitive to processor failures. This is because now at any moment a breakdown could cause loss of crucial information, while originally this only was the case for connections which happened to be open. However, depending on the communication environment, this might be worth the advantage of a 3-way handshake instead of a 4-way handshake. We will show in the sequel how we can achieve correct communication in the absence of processor breakdowns with a 3-way handshake.

Secondly, we could be less strict about correctness. For example, we could let a failure close mean: *It is possible that no message came across* instead of *No message came across*, thus leaving it to the host of the processor to decide whether to try to send it across once more. This could introduce duplication on a higher level.

Thirdly, we could try to base the protocol on a different principle using something which is more or less common to all processors, such as time. An example of this are the timer-based protocols described by Fletcher and Watson [2]. The correctness proof in [6] however, shows that the communication is less reliable, in the sense described above.

The fourth way out could be to restrict the mistakes the communication network makes. For example, we still allow loss of packets, but it is assumed that communication over the network has the FIFO property. This almost amounts to defining the problem of connection management away.

8.1. Correct communication with a 3-way handshake.

Reconsider the proof of lemma 6.3 to see what the problem was in the old model. A processor open with $g=0$ and $a=1$ which receives an error packet, does not have enough information to decide whether to close with a success or with a failure. Hence error packets will contain one bit of information extra, and whether a processor closes as a success or as a failure will now depend on this bit too, not only on its value of a and g. Remember we allowed processors to maintain information about previous connections. Since failure closes in no way contribute to correct communication, it is perhaps not surprising that we need information about past successful closes. It turns out that it is enough for a processor to remember its *yci* value of its last successful close. The processor will maintain this value in the variable *lsc*. The information which the extra bit in error packets carries is whether the x-field in the invalid packet $<x,y,z>$ is equal to the value in *lsc* or not.

Thus in the modified protocol skeleton, consisting of operations S^3, R^3, and C^3, only the error procedure and the closing operation are changed. As nonarbitrary closes still are necessary for correct communication, we incorporate this in C^3.

```
$S_P^3$:  same as $S_P$

$R_P^3$:  same as $R_P$, but with procedure error as follows:
      procedure error($<x,y,z>$) =
      begin if $lsc_P = x$ then $b$ := true else $b$ := false;
            if connection with Q closed then send <error,$y,x,b$>
            else if (($x = yci_P$ or $a_P = 0$) and ($y = mci_P$)) then skip
            else send <error,$y,x,b$>
      end

$C_P^3$:  if connection with Q is open then
      if <error,$yci_P, mci_P, b$> received or $a_P \geq last_P$ then
      begin if $a_P \leq cf_P$ then report failure
            else if $a_P \geq cs_P$ then begin report success; $lsc_P := yci_P$ end
            else if $b$ = true then begin report success; $lsc_P := yci_P$ end
            else report failure;
            $mci_P$:=undefined; $yci_P$:=undefined; $a_P$:=undefined; $g_P$:=undefined
      end
```

where

$last_P$	equal $2-g_P$,
cs_P	equal $2-g_P$,
cf_P	equal 0,
lsc_P	initially is undefined (distinct from 0), as long as no successful close has occurred.

Lemma 8.1. Let P and Q use the protocol skeleton consisting of operations S^3, R^3, and C^3.

(1) Let the following two conditions hold:

  (1) connections do not stay open indefinitely,

  (2) there are no processor breakdowns.

  Then $closed_P(m,y,g,a) \Rightarrow ((sclosed_P(m,y,g) \wedge a+g \geq 2) \vee eclosed_P(m,y,g,a))$.

(2) $a_P = 0 \Rightarrow (\neg\, sclosed_Q(m, mci_P, g) \wedge lsc_Q \neq mci_P)$.

(3) $a_P \geq 1 \Rightarrow (sclosed_Q(yci_P, mci_P, g) \Leftrightarrow lsc_Q = mci_P)$.

Proof. (1). Obvious from the protocol skeleton.

(2). Let $a_P = 0$ and $sclosed_Q(m, mci_P, g)$. Let this close be an (m, mci_P, g, a) close. Then there are two cases (lemma 8.1 (1)).

Case 1. $a+g \geq 2$. Hence with lemma 3.5, $a_P \geq 1$. Contradiction.
Case 2. $eclosed_Q(m, mci_P, 0, 1)$. This implies $closed_P(mci_P, y', g', a')$ (lemma 5.2). Contradiction.
Thus $a_P=0$ implies $\neg sclosed_Q(m, mci_P, g)$. Hence also $lsc_Q \neq mci_P$.
(3). The relation holds initially. S_P changes mci_P if a connection is opened, however then $a_P=0$ and thus $\neg sclosed_Q(m, mci_P, g)$ and $lsc_Q \neq mci_P$ hold. If R_P opens a connection, Q cannot know mci_P yet, thus $\neg sclosed_Q(m, mci_P, g)$ and $lsc_Q \neq mci_P$ hold. If R_P sets a_P from 0 to 1, we also have $\neg sclosed_Q(m, mci_P, g)$ and $lsc_Q \neq mci_P$, because of (2). Other changes of a_P in R_P do not affect the relation. C_P invalidates $a_P \geq 1$, and R_Q and S_Q do not change the relation in any way. Consider the operation C_Q. Let it result in an (m', y', g', a') close. If it is a failure close, then the relation is not affected. Let the close be successful. Let $a_P \geq 1$ hold (otherwise the relation holds trivially). We have 4 cases.
Case 1. $m' = yci_P \wedge y' = mci_P$. Then $sclosed_Q(yci_P, mci_P, g)$ becomes true and lsc_Q is set to mci_P.
Case 2. $m' = yci_P \wedge y' \neq mci_P$. Then $sclosed_Q(yci_P, mci_P, g)$ did not hold, nor $lsc_Q = mci_P$. Neither holds after this close.
Case 3. $m' \neq yci_P \wedge y' = mci_P$. Hence (lemma 3.7), $a'=1$ and $g'=0$. Thus it was an error close. However, the corresponding relation for Q also held before this operation C_Q: $a_Q \geq 1 \Rightarrow (sclosed_P(yci_Q, mci_Q, g) \Leftrightarrow lsc_P = mci_Q)$. Hence $sclosed_P(yci_Q, mci_Q, g)$ holds. Since $yci_Q = y' = mci_P$, we have a contradiction (P has not closed this connection yet). Hence this case cannot occur.
Case 4. $m' \neq yci_P \wedge y' \neq mci_P$. If $lsc_Q \neq mci_P$ before the close, the relation holds afterwards too. Assume $lsc_Q = mci_P$. Hence $sclosed_Q(yci_P, mci_P, g)$ holds. Hence (lemma 3.6), $a_Q + g_Q = 1$, and the close must be an error close. We also have $\neg\, closed_P(m'', mci_Q, g'', a'')$, hence $lsc_Q \neq mci_P$. Thus this close cannot be successful. Contradiction.
Summarizing, the relation still holds after an operation C_Q, which completes the proof. ■

Theorem 8.2. Let P and Q use the protocol skeleton consisting of operations S^3, R^3, and C^3. Let the following two conditions hold:
(1) connections do not stay open indefinitely,
(2) there are no processor breakdowns.
Then P does a successful (m, y, g) close iff
    Q does a successful (y, m, g') close.

Proof. Let P do a successful (m, y, g) close, and let it be an (m, y, g, a) close. According to lemma 8.1 we have two cases.
Case 1. $a+g \geq 2$. Hence with lemma 3.5 $(mci_Q = y \wedge yci_Q = m)$ or $closed_Q(m, y, g', a')$. Using lemmas 8.1 and 5.3 we have that $closed_Q(m, y, g', a')$ implies $sclosed_Q(m, y, g', a')$. Let Q be open still with $mci_Q = y$. Then $a_Q \geq 1$ and hence (lemma 8.1 (3)), $sclosed_P(yci_Q, mci_Q, g) \Leftrightarrow lsc_P = mci_Q)$. Thus a subsequent error close of Q will be a successful (y, m, g') close.
Case 2. $a=1$, $g=0$, and $eclosed_P(m, y, g, a)$. Thus Q had $lsc_Q = mci_P$ when it sent the error packet to P. As $a_P = 1$, we know (lemma 8.1 (3)) that $sclosed_Q(yci_P, mci_P, g')$ holds. Hence a successful close by P implies that Q does a corresponding successful close. The reverse

implication can be proved by an analogous argument. ■

Corollary 8.3. For correct communication in the absence of processor breakdowns, a 3-way handshake is sufficient if processors are allowed to remember one piece of information about a previous connection.

We remark that, although it is not necessary for the correctness of the protocol skeleton to test whether an opening packet contains a y-field equal 0, it is necessary now for an efficient protocol. This is because, in case the last packet of the 3-way handshake is lost, a processor needs an error reply on its packet $<x,y,0>$ in order to close gracefully. The chance to get that is small if the other processor (because it already closed) reopens on receipt of $<x,y,0>$.

Of course this modified protocol skeleton also can be extended to handle a multiple-message exchange. It makes however no sense to do so, since to transmit n messages we still need $last \geq n+2$.

8.2. Less reliable communication.

If we are willing to accept communications which are not always correct, the question still is how far we want to go. One possibility is to accept duplication and just require that there is no loss. In that case a 2-way handshake is sufficient to achieve our goal.

Theorem 8.4. Let $cf=0$, $cs=1$ and $last=1$ in the protocol skeleton consisting of operations S, R, and C. Let connections always be closed eventually. Then we do not have the problem of loss, i.e.

P does a successful $(m,y,1)$ close implies
Q does a successful (m',y',g') close with $y'=m$.

Proof. Since P did a successful $(m,y,1)$ close, $a_P+g_P \geq 2$. Hence Q was open with $mci_Q=y$, $yci_Q=m$ and $a_Q=1$. Thus, when Q closes this connection, it does a successful (y,m,g') close. ■

There is however a notion of correctness which lies somewhere in between "no loss" and "correct communication". We will call this *semi-correct communication*, and it is weaker than correct communication because it does not avoid all duplications. In the literature, *semi-correct* is sometimes called *correct.*

Definition 8.1. *Semi-correct communication* is the situation in which
P does a successful $(m,y,1)$ close implies
Q does exactly one successful (m',y',g') close with $y'=m$.

Note that semi-correct communication is strictly weaker than correct communication, because it allows that P does a failure $(m,y,1)$ close while Q does a successful $(y,m,0)$ close, which is excluded by correct communication.

Could we achieve semi-correct communication with a 2-way handshake? The problem is of course, how to avoid duplication in the case that the sender closes successfully. The receiver now cannot wait for the sender to tell it whether it made a duplicate opening, which is the way the problem was handled in the previous cases. This is because waiting for a packet from the sender turns the 2-way handshake into a 3-way handshake. One could argue that the receiver could just wait for an error packet, to tell how to close, but then we have the 3-way handshake discussed in section 8.1, where the sender just refuses to send his last packet with *seq*$=1$. Since the protocol skeleton does not require the sender to send this packet anyway, we even have correct communication, but we do not think it is fair to call this a 2-way handshake if we always (and not only in the case that a packet got lost) need a third (error) packet. Thus the receiver must decide on its own whether to open upon a packet and to risk a duplication, or not. Although if the receiver always refuses to open upon receipt of a packet this trivially satisfies the definition of semi-correct communication, we would hardly like to call this 'communication'. The receiver needs the *yci* value of all past successful closes in order to decide whether opening upon a received packet would introduce duplication or not, because this received packet could be a retransmission of a packet which led to a successful close arbitrarily long ago. We doubt that this is feasible in any practical case.

There is however a semi-correct 2-way handshake which needs only one piece of information from a previous connection, if we allow an additional assumption on the magic function *new value*. Namely, we require *new value* to be strictly increasing. This might seem a heavy condition to put on the function *new value*, but the two most obvious ways to implement this function both have this property. The first is, just to number any connections consecutively, and the other is, to use the current time. Although both implementations in theory use unbounded numbers, in practice this is not the case, since for example the number of milliseconds in two decades still fit in only 40 bits.

If a processor now receives a packet with an x-field less than or equal to the value it maintains in *lsc*, it knows it is an old retransmission and hence can discard it. Since successful closes are now possible with $a=0$ and hence $yci=0$, we should take care not to destroy *lsc* then. However, it turns out to be sufficient to remember the *yci*-value of the last successful close with $g=0$.

```
$S_P^4$:  same as $S_P$

$R_P^4$:  receive $<x,y,z>$;
          if connection with Q closed
          then  if $z \neq 0$ or $x \leq lsc_P$ then error $(<x,y,z>)$
                else begin $mci_P := new\ value$; $yci_P := x$; $g_P := 0$; $a_P := 1$ end      (*open*)
          else  if not (($x = yci_P$ or $a_P = 0$)    (*x valid*)
                        and $y = mci_P$               (*y valid*)
                        and $z = a_P$)                (*z valid*)
                then error $(<x,y,z>)$
                else begin $a_P := a_P + 1$; if $z = 0$ then $yci_P := x$ end
          with procedure error as in $R_P^3$.
```

```
$C_P^4$: if connection with Q is open then
    if <error, $yci_P, mci_P, b$> received or $a_P \geq last_P$ then
    begin if $a_P \leq cf_P$ then report failure
        else if $a_P \geq cs_P$ then begin report success; $lsc_P := yci_P$ end
        else if $b = $ true then begin report success; if $yci_P > 0$ then $lsc_P := yci_P$ end
        else report failure;
        $mci_P$:=undefined; $yci_P$:=undefined; $a_P$:=undefined; $g_P$:=undefined
    end
```

where

$last_P$	equal 1,
cs_P	equal 1,
cf_P	equal $0 - g_P$.

Lemma 8.5. Let P and Q use the protocol skeleton consisting of operations S^4, R^4, and C^4.

(1) Let the following two conditions hold:
   (1) connections do not stay open indefinitely,
   (2) there are no processor breakdowns.
   Then $closed_P(m,y,g,a) \Rightarrow ((sclosed_P(m,y,g) \wedge a \geq 1) \vee eclosed_P(m,y,g,a))$.

(2) lsc_P is increasing.

Proof. (1). Obvious from the protocol skeleton.
(2). Let $g_P = 0$. Then it is clear from the protocol skeleton that $yci_P > lsc_P$. Let $g_P = 1$ and $a_P \geq 1$. Consider the operation R_Q^4 in which Q opened the connection with $mci_Q = yci_P$. At that moment we had $mci_P > lsc_Q$, otherwise Q would not have opened this connection, and $lsc_P < mci_Q$, since mci_Q was put to a new value which is strictly greater than all previous values. Hence $yci_P > lsc_P$ as soon as P puts a_P to 1. Since P is open still and yci_P nor lsc_P are changed, $yci_P > lsc_P$ still holds. Since lsc_P is not changed with a successful close if $yci_P = 0$, lsc_P is increasing. ■

Theorem 8.6. Let P and Q use the protocol skeleton consisting of operations S^4, R^4, and C^4. Let the following two conditions hold:

(1) connections do not stay open indefinitely,
(2) there are no processor breakdowns.

Then P does a successful $(m,y,1)$ close implies
    Q does exactly one successful (m',y',g') close with $y' = m$.

Proof. Let P do a successful $(m,y,1)$ close. Let it be an $(m,y,1,a)$ close. There can be two cases (lemma 8.5).
Case 1. $a \geq 1$. Thus $a+g \geq 2$ and lemma 3.5 implies $closed_Q(y,m,0,a')$ with $a' \geq 1$, and hence a successful close, or $mci_Q = y$, $yci_Q = m$, and $a_Q \geq 1$, in which case Q's close will be successful, too.
Case 2. $a = 0$. Hence it is an error close, and $sclosed_Q(m',m,g')$ holds.
Thus in both cases Q does a successful close. Assume Q does two: an (m',m,g') close and an (m'',m,g'') close. With lemma 3.7 we have that $g' = g'' = 0$. Let the close with $mci_Q = m''$ be the last one. Hence just before the close $yci_Q > lsc_Q$. Since at the other close yci_Q was set

to m, and lsc_Q is increasing, we also have $lsc_Q \geq m$ just before the close with $mci_Q = m''$. Contradiction. Thus Q does exactly one successful close. ■

Corollary 8.7. For semi-correct communication in the absence of processor breakdowns, a 2-way handshake is sufficient if processors are allowed to remember one piece of information about a previous connection and the function *new value* is strictly increasing.

Again, for an efficient working protocol we need to test whether $y=0$ in an opening packet. Further, we can refrain from sending error packets if $x<lsc$. Since in a 2-way handshake the *seq*-field is 0 always, we can just as well leave this field out too in an implementation.

Note that the protocol skeleton contains a feature that is not necessary for its semi-correctness. Namely, if a processor with $g=1$ and $a=0$ receives an error packet, the way it closes depends on the bit in the error packet. This is not necessary since the processor could close as a failure always, even if the other one closed as a success. This still complies with the definition of semi-correctness. This suggests two ways to change the protocol skeleton.

Firstly, let a processor with $g=1$ and $a=0$ always close as a failure on receipt of an error packet. Then we do not need the extra bit in the error packets, nor the nonarbitrary closes, and hence we could refrain from sending error packets altogether. Furthermore, the *mci*-value of a processor with $g=0$ is never used, and only the processor with $g=0$ needs to remember lsc, to prevent duplication.

Secondly, consider why a processor with $g=1$ would want to close with $a=0$, apart from the case that the acknowledgement was lost (this case is handled correctly with use of the bit in the error packet). Originally, this was necessary to prevent deadlock if both processors happened to open simultaneously. If we can exclude this possibility, we have a correct 2-way handshake. The obvious way to do so, is to let the processors have a fixed g, say P is always sender ($g=1$) and Q is always receiver ($g=0$). In general, this clearly is not acceptable, but it might be inherent to some special application that the message transport is one way.

Theorem 8.8. Let P and Q use the protocol skeleton consisting of operations S^4, R^4, and C^4. Let the following three conditions hold:

(1) connections do not stay open indefinitely,
(2) there are no processor breakdowns,
(3) $g_P=1$ and $g_Q=0$ always.

Then P does a successful $(m,y,1)$ close iff
Q does a successful $(m',m,0)$ close.

Proof. Left to the reader. ■

Although the equivalence in theorem 8.8 is not sufficient to be correct communication by definition (P might close successfully with $yci_P=0$), we still feel this communication is correct (one processor closes successfully if and only if the other processor closes successfully).

Another question is, whether it is perhaps possible to have semi-correct communication with a 3-way handshake if we are not willing to use the extra memory necessary for a correct 3-way handshake. However, that we only have to avoid duplication when the sender closes successfully, turns out to be not much of a relaxation, as we can see from the proof of lemma 6.3. A receiver of an error packet which is in state $g=0$ and $a=1$ still does not have information enough to know how to close, and the information it needs, pertains to an already closed connection. Even the knowledge that *new value* is increasing is not enough if the processors are not allowed to remember the highest value seen, after a connection is closed.

8.3. The use of time. The assumptions on which timer-based protocols for connection management rely differ in two aspects from our original assumptions. The first one is that packets do not have arbitrary delays: there is some finite time after which we can assume that a packet is lost, if it has not arrived yet. The second one is that processors can use time: they have local clocks, and the local clocks are in some way related. The way the first assumption usually is implemented, uses time, too. Messages are timestamped when sent, and if any processor encounters a packet with a timestamp which is older than some fixed lifetime, it is destroyed. Thus, if the communication network allows arbitrary delays, we need that the local clocks all show more or less the same time. That is, the processors' clocks may drift away from real time, but the drift of all clocks should remain ρ-bounded (see e.g. [6]). The way duplication is avoided in timer-based protocols is basicly the same as in the 2-way handshake from the previous subsection. Do not open a connection when it is a duplication. More specifically, do not open a connection upon an old packet. In the previous, 'old' meant: with an x-field less than or equal to lsc, now 'old' means: with a timestamp $\leq$ local time - packet lifetime. The way this is handled is that connections are left open until no more packets from the current connection can arrive. However if, once a connection is opened, processors have to be able to conclude that no more packets of the current connection can arrive, processors also must agree upon sending retransmissions for a fixed time only. This has one serious drawback: if packets can be sent only for a fixed time, all these packets might be lost in the communication network. In all previous protocol skeletons we had the possibility to just continue trying to get through and send retransmissions until some answer is received, whether it is the expected answer or an error packet. This is the reason why timer-based protocols only achieve semi-correct communication. As an illustration we give the protocol skeleton for a timer-based semi-correct 2-way handshake for single-message communication, leaving out all features that facilitate multiple-message flow as in e.g. the protocol given by Fletcher and Watson [2]. We incorporate the feature of destroying outdated packets in the protocol skeleton to comply with the assumption of unbounded delays. Apart from the atomic operations S^5, R^5, and C^5 for both processors, we add an atomic action T (time) which increases the local clocks of both processors with the same amount τ. This can be interpreted as: during time τ the processors did no S^5, R^5, or C^5 operations. The idea of the atomic action time is due to Tel [6]. Thus we assume the local clocks show exactly the same time. It is easy to adapt the protocol skeleton to clocks which have a drift which is ρ-bounded, see e.g. [6]. The problem of how to keep local clocks in a distributed system synchronized is nontrivial, but lies outside the scope of this paper. In this case it is necessary for the semi-correctness of the protocol skeleton that a processor can distinguish opening packets from acknowledgements.

```
T:  choose $\tau \in \mathbb{R}^+$;
    $clock_P := clock_P + \tau$; $clock_Q := clock_Q + \tau$
```

```
$S_P^5$:  if connection with Q closed then
          begin $mci_P := clock_P$; $a_P := 0$; $g_P := 1$ end;
          if $clock_P - mci_P < mst_{g_P}$
          then send a packet $<seq, clock_P, D>$ where $seq < a_P + g_P$

$R_P^5$:  receive $<z, t, d>$;
          if $clock_P - t < mpl$ then
          begin if connection with Q closed
                then  if $z \neq 0$ or $d = ack$ then error $(<z,t,d>)$
                      else begin $mci_P := clock_P$; $D := ack$; $g_P := 0$; $a_P := 1$ end      (*open*)
                else  if not ((($g_P = 1$ and $d = ack$) or
                             ($g_P = 0$ and $d \neq ack$))
                             and $a_P = z$)
                      then error $(<z,t,d>)$
                      else $a_P := a_P + 1$
          end
          where procedure error could be 'skip'.

$C_P^5$:  if connection with Q is open and $clock_P - mci_P \geq mct_{g_P}$ then
          begin if $a_P \geq cs_P$ then report success else report failure;
                $mci_P$:=undefined; $a_P$:=undefined; $g_P$:=undefined
          end
```

where

mci_P	now contains the time the current connection was opened. (yci_P will not be used any more.)
cs_P	is 1 for the 2-way handshake,
mst_{g_P}	is the maximum time during which a packet may be sent. It may depend on the value of g_P.
mpl	is the maximum time during which a packet can live in the communication network,
mct_{g_P}	is the minimum connection time (to prevent duplication). It may depend on the value of g_P.

Clearly the correctness of the protocol skeleton depends on the way the constants mst_0, mst_1, mpl, mct_0, and mct_1 are chosen.

Although the assumptions on which this timer-based protocol skeleton is based are different from those of the previous protocol skeletons, we will show that it is a restriction of protocol skeleton 4 from section 8.2. For this purpose we define a timer-based protocol skeleton consisting of operations T, $S^{5'}$, $R^{5'}$, and $C^{5'}$, where the variables *yci* and *lsc* and the packet fields x and y are added to protocol skeleton 5. Thus we can compare situations in the protocol skeletons 4 and 5', and hence in 4 and 5. We will add the number of the protocol skeleton as a superscript if we need to make this distinction. For example, we will show that $valid^{5'}(<x,y,z,t,d>)$ implies $valid^4(<x,y,z>)$.

```
T:      choose $\tau \in \mathbf{R}^+$;
        $clock_P := clock_P + \tau$; $clock_Q := clock_Q + \tau$

$S_P^{5'}$:  if connection with Q closed then
        begin $mci_P := clock_P$; $a_P := 0$; $g_P := 1$; $yci_P := 0$ end;
        if $clock_P - mci_P < mst_{g_P}$
        then send a packet $<mci_P, yci_P, seq, clock_P, D>$ where $seq < a_P + g_P$

$R_P^{5'}$:  receive $<x,y,z,t,d>$;
        if $clock_P - t < mpl$ then
        begin if connection with Q closed
              then  if $z \neq 0$ or $d = ack$ then $error$ $(<x,y,z,t,d>)$
                    else begin $mci_P := clock_P$; $D := ack$; $g_P := 0$; $a_P := 1$; $yci_P := x$ end (*open*)
              else  if not $(((g_P = 1$ and $d = ack)$ or
                           $(g_P = 0$ and $d \neq ack))$
                           and $a_P = z)$
                    then $error$ $(<x,y,z,t,d>)$
                    else begin $a_P := a_P + 1$; if $z = 0$ then $yci_P := x$ end
        end
        where procedure $error$ could be 'skip'.

$C_P^{5'}$:  if connection with Q is open and $clock_P - mci_P \geq mct_{g_P}$ then
        begin if $a_P \geq cs_P$ then begin $lsc_P := yci_P$; report success end
              else report failure;
              $mci_P$:=undefined; $a_P$:=undefined; $g_P$:=undefined; $yci_P$:=undefined
        end
```

Lemma 8.9. Let P and Q operate with the protocol skeleton consisting of operations T, $S^{5'}$, $R^{5'}$, and $C^{5'}$. Then

(1) lemmas 3.1, 3.2 (1)-(3), and 3.3 (1) hold,

(2) $clock_P = clock_Q$,

(3) $closed_P(m,y,g,a) \Rightarrow (clock_P - m \geq mct_g \wedge (mci_P \neq \text{undefined} \Rightarrow mci_P - m \geq mct_g))$,

(4) $lsc_P \neq \text{undefined} \Rightarrow closed_P(m, lsc_P, g, a)$,

(5) $sent_P(<x,y,z,t,ack>) \Rightarrow (0 \leq t - x < mst_0 \wedge ((mci_P = x \wedge g_P = 0) \vee closed_P(x,y,0,a)))$,

(6) $sent_P(<x,y,z,t,d \neq ack>) \Rightarrow (0 \leq t - x < mst_1 \wedge ((mci_P = x \wedge g_P = 1) \vee closed_P(x,y',1,a)))$,

(7) $((mci_P = x \wedge g_P = 0) \vee closed_P(x,y,0,a)) \Rightarrow$
$((\neg sent_P(<x',y',z,t,ack>) \wedge x + mst_0 \leq t < x + mct_0) \wedge$
$(\neg sent_P(<x',y',z,t,d \neq ack>) \wedge x \leq t < x + mct_0)$,

(8) $((mci_P = x \wedge g_P = 1) \vee closed_P(x,y,1,a)) \Rightarrow$
$((\neg sent_P(<x',y',z,t,ack>) \wedge x \leq t < x + mct_1) \wedge$
$(\neg sent_P(<x',y',z,t,d \neq ack>) \wedge x + mst_1 \leq t < x + mct_1)$.

Proof. Directly from the protocol skeleton. ■

Lemma 8.10. Let P and Q operate with the protocol skeleton consisting of operations T, $S^{5'}$, $R^{5'}$, and $C^{5'}$. Let $mct_1 > 2mpl + mst_0 + mst_1$ and $mct_0 > mpl + mst_1$. Then

(1) $g_P = 0 \Rightarrow 0 \le mci_P - yci_P < mpl + mst_1$,

(2) $(g_P = 1 \wedge a_P \ge 1) \Rightarrow 0 \le yci_P - mci_P < mpl + mst_1$,

(3) $(mci_P = \text{undefined} \wedge lsc_P \ne \text{undefined} \wedge sent_Q(<x,0,0,t,d>) \wedge clock_P - t < mpl) \Rightarrow x > lsc_P$,

(4) $(mci_P \ne \text{undefined} \wedge sent_Q(<x,y,z,t,d>) \wedge valid_P^{5'}(<x,y,z,t,d>)) \Rightarrow y = mci_P$,

(5) $(closed_P(m,y,g,a) \wedge a \ge 1 \wedge a_P \ge 1) \Rightarrow yci_P \ne y$,

(6) $(sent_Q(<x,y,z,t,d>) \wedge valid_P^{5'}(<x,y,z,t,d>)) \Rightarrow valid_P^4(<x,y,z>)$.

Proof. (1). Use lemma 8.9 (6) and the fact that the packet upon which P opened the connection contained a t-field which was tested for $clock_P - t < mpl$.

(2). Use lemmas 8.9 (5) and 8.10 (1).

(3). $lsc_P \ne$undefined implies $closed_P(m,y,g,a)$ with $y = lsc_P$ and hence $clock_P - m > mct_g$. We have two cases.

Case 1. $g = 1$. With lemma 8.10 (1) we have $clock_P - lsc_P > mct_1 - mpl - mst_1$. Using lemma 8.9 (7) for Q we have that $t \ge x > lsc_P + mct_0$, which implies $x > lsc_P$, or that $t < lsc_P$. Using lemmas 8.9 (3) and (7), we have that $t < lsc_P - mct_1 + mct_1$. This is a contradiction with $clock_P - t < mpl$ and $clock_P - lsc_P > mct_1 - mpl - mst_1$.

Case 2. $g = 0$. With lemma 8.10 (2) we have $clock_P - lsc_P > mct_0$. Using lemma 8.9 (8) for Q we have that $t \ge x > lsc_P + mct_1$, which implies $x > lsc_P$, or that $t < lsc_P + mst_1$. This is a contradiction with $clock_P - t < mpl$ and $clock_P - lsc_P > mct_0$.

(4). We have the following cases.

Case 1. $g_P = 0$. Hence $d \ne ack$ and since $a_P \ge 1$, $z \ge 1$ and $y \ne 0$. With lemmas 8.9 (6) and 8.10 (2) we have $0 \le clock_P - y < mpl + mst_1$. Since $clock_P - mci_P \ge 0$, we have $mci_P - y < mpl + mst_1 < mct_0$. Hence $mci_P - y \ge 0$ implies $mci_P = y$. Assume $mci_P < y$. Since P has not closed the connection between mci_P and $clock_P$, $mci_P(y) = mci_P = y$.

Case 2. $g_P = 1$. Hence $d = ack$ and with lemmas 8.9 (5) and 8.10 (1) we have $mci_P - y < 2mpl + mst_0 + mst_1 < mct_1$. Thus for the same reasons as in the first case, $mci_P = y$.

(5). Assume first that $yci_P = y$. Then $mci_Q = y$ or $closed_Q(y, y', g', a')$. This single connection had only one value of g_Q or g', respectively. Hence the case that $g \ne g_P$ cannot occur. Thus we have two cases left.

Case 1. $g = g_P = 0$. Hence $mci_P - m > mct_0$. With lemma 8.10 (1) and $mct_0 > mpl + mst_1$ this leads to $yci_P > y$.

Case 2. $g = g_P = 1$. Hence $mci_P - m > mct_1$. With lemma 8.10 (2) and $mct_1 > mpl + mst_1$ this leads to $yci_P > y$.

(6). Note that lemma 8.10 (3) ensures that $sent_Q(<x,0,0,t,d>)$ and $valid_P^{5'}(<x,0,0,t,d>)$ imply $valid_P^4(<x,0,0>)$. All other packets are only $valid_P^{5'}$ if $mci_P \ne$undefined. Hence we can use lemma 8.10 (4). Since yci_P-values unequal 0 uniquely identify connections (lemma 8.10 (5)), a valid y-value implies a valid x-value. The test whether a z-value is valid is the same in both protocol skeletons, thus we have proved the conclusion. ■

Theorem 8.11. The protocol skeleton consisting of operations T, S^5, R^5, and C^5 is a restriction of the protocol skeleton consisting of operations S^4, R^4, and C^4.

Proof. The only difference between the protocol skeletons 5 and 5' is that in 5 all those packet fields and variables which are not used in 5', are discarded. Since acceptance of a packet in $R^{5'}$ implies acceptance of that packet in operation R^4, $R^{5'}$ is a restriction of R^4. The operation $S^{5'}$ contains an extra restriction upon sending compared to S^4, hence operation $S^{5'}$ is a restriction of S^4. Since we saw in the discussion of section 8.2 that sending error packets and using the information therein for closing is not essential, and that always closing as a failure in case $a=0$ and $g=1$ yields a semi-correct protocol skeleton, too, operation $C^{5'}$ is a restriction of C^4. Since operation T is a restriction of the function *new value*, protocol skeleton 5 is a restriction of protocol skeleton 4. ■

Corollary 8.12. For semi-correct communication in the absence of processor breakdowns, a 2-way handshake is sufficient if processors have access to synchronized local clocks.

We remark that although the straightforward extension of this protocol skeleton for multiple-message communication as shown in section 7 works, it will greatly improve if there is not just a constant time available to send the sequence of messages, but some time dependent on the length of the sequence. Hence in the timer-based protocol of Fletcher and Watson [2] each message in the sequence has its own timer. For a partial correctness proof of this protocol which also uses system-wide invariants, we refer to [6].

9. References.

[1] Belsnes, D., *Single-Message Communication*, IEEE Trans. Commun. 24 (1976), 190-194.

[2] Fletcher, J.G. and R.W. Watson, *Mechanisms for a Reliable Timer-Based Protocol*, Computer Networks 2 (1978), 271-290.

[3] Knuth, D.E., *Verification of Link-Level Protocols*, BIT 21 (1981), 31-36.

[4] Krogdahl, S., *Verification of a Class of Link-Level Protocols*, BIT 18 (1978), 436-448.

[5] Schoone, A.A. and J. van Leeuwen, *Verification of Balanced Link-Level Protocols*, Techn. Rep. RUU-CS-85-12, Dept. of Computer Science, University of Utrecht, Utrecht, 1985. (Submitted for publication.)

[6] Tel, G., *Assertional Verification of a Timer-Based Protocol*, Techn. Rep. RUU-CS-87-15, Dept. of Computer Science, University of Utrecht, Utrecht, 1987.

[7] Tomlinson, R.S., *Selecting Sequence Numbers*, Proc. ACM SIGCOMM/SIGOPS Interprocess Commun. Workshop, ACM, pp. 11-23, 1975.