JAN 173 -0

The sliding window protocol revisited

Jan L.A. van de Snepscheut
Computer Science
California Institute of Technology
Pasadena, CA 91125

Summary

We give a correctness proof of the sliding window protocol. Both safety and liveness properties are addressed. We show how faulty channels can be represented as nondeterministic programs. The correctness proof is given as a sequence of correctness-preserving transformations of a sequential program that satisfies the original specification, with the exception that it does not have any faulty channels. We work as long as possible with a sequential program, although the transformation steps are guided by the aim of going to a distributed program. The final transformation steps consist in distributing the actions of the sequential program over a number of processes.

Key words

communication protocols, program transformation, guarded commands, fairness.

1 Introduction

In this note, we give a correctness proof of the sliding window protocol. We discuss both safety and liveness properties. We specialize our program to window size 1 and obtain the alternating bit protocol. The alternating bit protocol can be traced back to [2]. We have been unable to trace back the origins of the sliding window protocols; [7] discusses one of the versions and lists networks using related protocols.

2 A faulty channel

A communication protocol is used to provide reliable transmission of data over a faulty communication channel that garbles, duplicates, or loses data. We consider the case in which data is transmitted in one direction over the faulty channel, and we assume the presence of a channel in the opposite direction in order to be able to communicate the need for retransmission of a message. The latter channel is also faulty. No assumptions on the slack of the faulty or of the fault-free channel are to be made. It is assumed that a faulty channel operates as follows:

- messages arrive in the order in which they are sent;
- any message sent along the channel can be lost;
- any message sent along the channel can be duplicated;
- any message sent along the channel can be garbled; however, if a message is garbled this can be detected, i.e. the error detection mechanism is assumed to be perfect.
- the channel is not infinitely faulty, in the sense that only a finite number of messages can be lost or duplicated consecutively, and of the messages delivered only a finite number are garbled consecutively.

First, we give a program that implements a faulty channel. The program has input channel c and output channel d. The output on d is a faulty copy of the input on c, i.e. every message in d is accompanied by a boolean which indicates whether the message is garbled. One would use some form of coding and decoding to implement this boolean. We use functions *flip* and *flip′* which return a boolean value. They make a fair (but not necessarily random) choice between *true* and *false*. Channels c and d are fault free.

$$*[c?x;\ \ *[\mathit{flip} \rightarrow d!(x, \mathit{flip}')]]$$

If the inner loop is iterated zero times then a message is lost; if it is iterated more than once then a message is duplicated. The *flip′* that occurs as an argument with x in the output command corresponds to the possibility of garbling the message. The choices made by *flip* and *flip′* are independent, both of them are fair.

Second, we show that we can restrict ourselves to loss and duplication of messages, i.e. we may assume that no messages are garbled. The reception of a garbled message does not give any information at the receiving side; it cannot even be concluded that a new message was sent. Hence, the only sensible thing that can be done with a garbled message is to ignore it. Therefore, we propose to use the faulty channel only in conjunction with a program that filters out all garbled messages.

$$\begin{array}{ll} & *[c?x;\ \ *[\mathit{flip} \rightarrow d!(x, \mathit{flip}')]] \\ \| & *[d?(y, b);\ [b \rightarrow \mathit{skip} [\!] \neg b \rightarrow e!y]] \end{array}$$

The latter combination is equivalent to

$$*[c?x;\ \ *[\mathit{flip} \rightarrow e!x]]$$

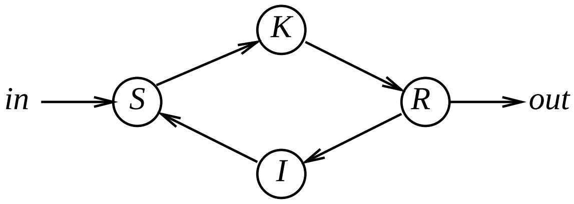
except for the slack in the communication, which is something that we want to ignore anyway. This program can also be written as

$$c?x;$$
$$*[true \rightarrow c?x \,[\!]\, true \rightarrow e!x]$$

where the choice between the two alternatives is assumed to be fair. This is the version of the faulty channel that we work with. It has the advantage that garbling of messages plays no role. The symmetry in input and output, apart from initialization, is also pleasing. The program for the faulty channel is similar but not identical to the faulty channel discussed in [4].

3 The sliding window protocol

In this section we consider the sliding window protocol. We have a network as indicated in the figure below. Channels K and I are faulty channels, and the task is to construct programs S and R that, together, implement a fault-free communication line from in to out. Both in and out are fault-free channels. We first consider a solution in which messages are numbered from 0 on and subsequently refine the program to bounded sequence numbers. It is easier to do it this way rather than starting with the latter program because we would then have to introduce the unbounded integers anyway for proving the program's correctness.



The problem we try to solve is to come up with a program that guarantees that the sequence of values transmitted via out is a prefix of the sequence of values transmitted via in. The difference in length is at most N, where N is a given positive constant called the window size. The program is embedded in a context in which input and output operations are always performed eventually.

We start with a sequential program that implements the specification and refine it later to a distributed program. The refinement is done in such a way that the distributed program consists of four processes, two of which are the faulty channels K and I. The sequential program uses variables n and j whose values equal the number of completed input and output operations. The essential invariant is

$$n = \#in \;\wedge\; j = \#out \;\wedge\; j \le n \le j + N \;\wedge$$
$$\langle \forall h : 0 \le h < n : a(h) = in_h \rangle \;\wedge\; \langle \forall h : 0 \le h < j : a(h) = out_h \rangle$$

in which c_h is the h th value communicated via channel c. The program performs two actions

$$in?a(n);\;\; n := n + 1$$
$$out!a(j);\;\; j := j + 1$$

while maintaining the above invariant. A program that does just this is

$$
\begin{array}{l}
n, j := 0, 0; \\
*[\ n \neq j + N \quad \wedge \quad \overline{in} \quad \rightarrow in?a(n);\ \ n := n + 1 \\
\ \ [\!]\ j \neq n \quad \wedge \quad \overline{out} \qquad \rightarrow out!a(j);\ \ j := j + 1 \\
\ \ ] \qquad .
\end{array}
$$

We are going to refine this program until we have a distributed version in which we can identify four processes, connected as shown in the picture. To that end, we partition the variables and assign each of them to a process. A requirement is that each action operate on the local variables of one process only. For example, variable n, which stores the number of $in?$ operations, will be assigned to process S because channel in is connected to S. Similarly, j is assigned to R. As a result, no process stores both n and j, and the therefore guards $n \neq j + N$ and $j \neq n$ have to be changed. Both the commands and the guards are going to change, but we deal with them one after the other. Whenever we introduce new variables, we capture their "intended meaning" in an invariant, and we propose a command that updates the new variable. From the invariant we then calculate the guard.

We focus on the sequence numbers first, and therefore ignore the $in?a(n)$ and $out!a(j)$ operations for a while. We are going to introduce some fresh variables. The choice of variables is partly based on the requirement that the variables can be partitioned, and partly on the fact that faulty channels have to be used. The latter forces weaker invariants than would have been the case with perfectly reliable channels.

One piece of notation. We use $\{i, ..j\}$ to denote a set of integers with the property

$$x \in \{i, ..j\} \ \equiv\ i \leq x < j \qquad .$$

It should come as no surprise that the asymmetry between i and j turns out to be helpful.

The first variable that we introduce is r. It is going to be a local variable of R and serves to eliminate variable n from the guard of action $j := j + 1$. The interpretation is

r is the set of sequence numbers received by R .

We choose a set of sequence numbers instead of just the number of messages because we anticipate that messages may be lost in the communication from S to R and therefore the sequence numbers received need not be consecutive anymore. Since j is the number of messages transmitted by R via out, and since those messages were first received via in and are numbered from 0 up to and excluding n, we have invariant

$$\{0, ..j\} \subseteq r \subseteq \{0, ..n\} \qquad .$$

We need to add a guarded command to update r. Letting k be

k is the number of the last message sent by S

we find that we need to add

$$r := r \cup \{k\} \qquad ,$$

to express receipt of a message from S by R. The invariant for k is

$$0 \leq k < n \qquad .$$

We need to add a command that assigns a new value to k. It is tempting to assign to k a value from the set $\{0,..n\} \setminus r$ since that is the set of numbers input by S and not yet received by R. However, this expression involves variables from both S (namely n) and R (namely r) and can therefore not be partitioned. We introduce variable s, which is meant to be be a copy of r but it might not be up to date. The latter is expressed by

$$s \subseteq r$$

but we will have to strengthen this a bit later. The command that we add to our program is

$$k := any\{0,..n\} \setminus s$$

which assigns to k an arbitrary element of $\{0,..n\} \setminus s$. It is well-defined only if such an element exists. Notice that no fair choice is specified here. The next command that we need to introduce is to update our newly introduced variable s. It would be nice to write $s := r$ but this is not feasible because the communication between S and R is limited to communication via faulty channels. Therefore, we introduce intermediate variable i whose value is a copy of r, but it might not be up to date. We say "intermediate" because its value will be somewhere in between s and r. (s is a possibly-out-of-date copy of i, and i is a possibly-out-of-date copy of r.)

$$s \subseteq i \subseteq r$$

This is the slightly stronger invariant referred to above. The commands for updating s and i are immediate.

$$s := i$$

$$i := r$$

It remains to introduce a local variable of S that tracks j. We may introduce a variable that is a copy of j, but not necessarily up to date. However, we already have such an approximation of j in the form of the smallest number missing from set s. We introduce $\overline{s}$ as an abbreviation for $\{0,..\infty\} \setminus s$ and introduce variable l.

$$l = \min(\overline{s})$$

Collecting all the terms of the invariant, we have

$$P: \ l = \min(\overline{s}) \leq \min(\overline{i}) \leq j \leq n \leq l + N \quad \wedge \quad 0 \leq k < n \quad \wedge \quad s \subseteq i \subseteq r \subseteq \{0,..n\} \quad \wedge \quad \{0,..j\} \subseteq r$$

in which the term $\min(\overline{s}) \leq \min(\overline{i})$ is redundant because it follows from $s \subseteq i$. We now calculate the guards of the six commands. The first command is $n := n + 1$, and

$$
\begin{array}{ll}
 & P^n_{n+1} \\
= & \\
 & l=\min(\overline{s})\le\min(\overline{i})\le j\le n+1\le l+N \quad\wedge\quad 0\le k<n+1 \quad\wedge \\
 & s\subseteq i\subseteq r\subseteq\{0,..n+1\} \quad\wedge\quad \{0,..j\}\subseteq r \\
\Leftarrow & \\
 & P \quad\wedge\quad n\ne l+N
\end{array}
$$

suggests guard $n\ne l+N$. The second command is $j:=j+1$, and

$$
\begin{array}{ll}
 & P^j_{j+1} \\
= & \\
 & l=\min(\overline{s})\le\min(\overline{i})\le j+1\le n\le l+N \quad\wedge\quad 0\le k<n \quad\wedge \\
 & s\subseteq i\subseteq r\subseteq\{0,..n\} \quad\wedge\quad \{0,..j+1\}\subseteq r \\
\Leftarrow & \\
 & P \quad\wedge\quad j<n \quad\wedge\quad j\in r \\
= & \qquad \{\ P \ \Rightarrow\ r\subseteq\{0,..n\}\ \} \\
 & P \quad\wedge\quad j\in r
\end{array}
$$

suggests guard $j\in r$. The third command is $r:=r\cup\{k\}$, and

$$
\begin{array}{ll}
 & P^r_{r\cup\{k\}} \\
= & \\
 & l=\min(\overline{s})\le\min(\overline{i})\le j\le n\le l+N \quad\wedge\quad 0\le k<n \quad\wedge \\
 & s\subseteq i\subseteq r\cup\{k\}\subseteq\{0,..n\} \quad\wedge\quad \{0,..j\}\subseteq r\cup\{k\} \\
\Leftarrow & \\
 & P
\end{array}
$$

suggests guard *true*. The fourth command is $k:=any\{0,..n\}\setminus s$, and

$$
\begin{array}{ll}
 & P^k_{any\{0,..n\}\setminus s} \\
= & \\
 & l=\min(\overline{s})\le\min(\overline{i})\le j\le n\le l+N \quad\wedge\quad 0\le any\{0,..n\}\setminus s<n \quad\wedge \\
 & s\subseteq i\subseteq r\subseteq\{0,..n\} \quad\wedge\quad \{0,..j\}\subseteq r \quad\wedge\quad \{0,..n\}\setminus s\ne\emptyset \\
\Leftarrow & \\
 & P \quad\wedge\quad \{0,..n\}\setminus s\ne\emptyset
\end{array}
$$

suggests guard $\{0,..n\} \setminus s \neq \emptyset$. The fifth command is $s := i;\ l := \min(\overline{s})$, and

$$
\begin{array}{ll}
& (P^l_{\min(\overline{s})})^s_i \\
= & \\
& \min(\overline{i}) = \min(\overline{i}) \leq \min(\overline{i}) \leq j \leq n \leq \min(\overline{i}) + N \quad \wedge \quad 0 \leq k < n \quad \wedge \\
& i \subseteq i \subseteq r \subseteq \{0,..n\} \quad \wedge \quad \{0,..j\} \subseteq r \\
\Leftarrow & \\
& P
\end{array}
$$

suggests guard *true*. The sixth command is $i := r$, and

$$
\begin{array}{ll}
& P^i_r \\
= & \\
& l = \min(\overline{s}) \leq \min(\overline{r}) \leq j \leq n \leq l + N \quad \wedge \quad 0 \leq k < n \quad \wedge \\
& s \subseteq r \subseteq r \subseteq \{0,..n\} \quad \wedge \quad \{0,..j\} \subseteq r \\
\Leftarrow & \\
& P \quad \wedge \quad \min(\overline{r}) \leq j \\
\Leftarrow & \\
& P \quad \wedge \quad j \notin r
\end{array}
$$

suggests guard $j \notin r$. Thus, we obtain the complete program. In order to enable initialization, the program starts off with $n = 1$, which corresponds to performing an initial input action prior to executing the loop.

$$
\begin{array}{lll}
j, k, l, n, s, i, r := 0, 0, 0, 1, \emptyset, \emptyset, \emptyset; & & \\
* \, [\ n \neq l + N & \rightarrow n := n + 1 \\
\quad [\!] \ j \in r & \rightarrow j := j + 1 \\
\quad [\!] \ \mathit{true} & \rightarrow r := r \cup \{k\} \\
\quad [\!] \ \{0,..n\} \setminus s \neq \emptyset & \rightarrow k := \mathit{any}\{0,..n\} \setminus s \\
\quad [\!] \ \mathit{true} & \rightarrow s := i;\ \ l := \min(\overline{s}) \\
\quad [\!] \ j \notin r & \rightarrow i := r \\
\quad] &
\end{array}
$$

Next, we prove progress. We claim the following variant function.

$$(j + \#s + \#i + \#r + n,\ s \neq r \vee k \notin r)$$

The first component of the pair is an integer, the second component is a boolean. The ordering of the pair is lexicographic ordering, where $true > false$. We claim that no statement decreases the variant function. The verification is straightforward and is omitted. The more interesting part is to show that the variant function strictly increases from time to time. We show that, in every state, a guarded command exists whose guard is true and whose command strictly increases the variant function. Furthermore, we show that if another guarded command falsifies the guard, it does so while increasing the variant function. If we postulate a weak, fair choice between the guarded commands of the loop, progress follows.

- In state $s \neq i$, guarded command $true \rightarrow s := i$ increases the variant function; its guard is not falsified by any command.
- In state $j \in r$, guarded command $j \in r \rightarrow j := j + 1$ increases the variant function; its guard is not falsified by any other command.
- In state $i \neq r \wedge j \notin r$, guarded command $j \notin r \rightarrow i := r$ increases the variant function; its guard can be falsified by only one other command, viz. $r := r \cup \{k\}$ if $j = k \wedge k \notin r$, which also increases the variant function.
- In state $s = i = r \neq \{0, ..n\} \wedge k \in r$, guarded command $\{0, ..n\} \setminus s \neq \emptyset \rightarrow k := any\{0, ..n\} \setminus s$ increases the variant function; its guard is not falsified by any other command.
- In state $s = i = r \neq \{0, ..n\} \wedge k \notin r$, guarded command $true \rightarrow r := r \cup \{k\}$ increases the variant function; its guard is not falsified by any other command.
- In state $s = i = r = \{0, ..n\}$, guarded command $n \neq l + N \rightarrow n := n + 1$ increases the variant function; its guard is $true$ (since $N > 0$) and is not falsified by any other command.

The disjunction of the six conditions is $true$, implying that all cases have been covered.

4 The range of k and j

Next, we turn to the reduction of sequence numbers. We introduce a variable that plays a role in the proof only: set K that allows us to record the set of all possible values that might have been chosen for k. The program is extended with this variable as follows.

$$
\begin{array}{ll}
\multicolumn{2}{l}{j, k, l, n, s, i, r, K := 0, 0, 0, 1, \emptyset, \emptyset, \emptyset, \emptyset;} \\
*[\; n \neq l + N & \rightarrow n := n + 1 \\
\;[\!]\; j \in r & \rightarrow j := j + 1 \\
\;[\!]\; true & \rightarrow r := r \cup \{k\} \\
\;[\!]\; \{0, ..n\} \setminus s \neq \emptyset & \rightarrow k := any\{0, ..n\} \setminus s;\;\; K := s \\
\;[\!]\; true & \rightarrow s := i;\;\; l := \min(\overline{s}) \\
\;[\!]\; j \notin r & \rightarrow i := r \\
\;] &
\end{array}
$$

We postulate the following invariant in addition to the one we already have.

$$k \in \{0,..n\} \setminus K \quad \wedge \quad k < \min(\overline{K}) + N \quad \wedge \quad j \leq \min(\overline{K}) + N \quad \wedge$$
$$K \subseteq s \quad \wedge \quad r \subseteq \{0,..\min(\overline{K}) + N\}$$

The invariant is established through execution of the first statement (since $N > 0$). We omit the check that every assignment to one of the variables maintains the invariant. It should be noted that assignment $K := s$ does not decrease $\min(\overline{K})$ since we have $K \subseteq s$. From the new and the old invariant, we have

$$\min(\overline{K}) \leq j \leq \min(\overline{K}) + N \quad \wedge \quad \min(\overline{K}) \leq k < \min(\overline{K}) + N$$

from which we conclude

$$-N < j - k \leq +N \qquad ,$$

which allows sequence numbers to be reduced modulo $2N$ at the receiving side. Because of

$$l \leq n \leq l + N \qquad ,$$

reduction modulo $2N$ presents no problems at the sending side either.

We may also reduce the size of the various sets involved. Instead of the unbounded set r, we may introduce r' such that

$$r' \;=\; r \setminus \{0,..j\}$$

(because the invariant implies $\{0,..j\} \subseteq r$). Also, because the invariant implies $\min(\overline{s}) \leq n \leq \min(\overline{s}) + N$, we may introduce

$$s' \;=\; \overline{s} \cap \{\min(\overline{s}),..\min(\overline{s}) + N\}$$

and similarly

$$i' \;=\; \overline{i} \cap \{\min(\overline{i}),..\min(\overline{i}) + N\}$$

and work with sets that have at most N elements each.

If we add the original messages back into the system, we need to add variables to store some messages. The variables are a, b, and v and their use is governed by invariant

$$\langle \forall h : 0 \leq h < n : a(h) = in_h \rangle \quad \wedge \quad \langle \forall h : 0 \leq h < j : out_h = in_h \rangle \quad \wedge$$
$$\langle \forall h : h \in r : b(h) = in_h \rangle \quad \wedge \quad v = in_k \qquad .$$

Because of the various bounds that we have established, all of those variables store at most $2N$ messages. The program (in which we haven't done the reduction modulo $2N$) looks like this.

$$
\begin{array}{l}
j, l, n, s', r' := 0, 0, 1, \{0, ..N\}, \emptyset;\ \ in?a(0);\ \ k, v := 0, a(0);\ \ i' := s';\\
\ast\ [\ n \neq l + N \qquad\quad \rightarrow in?a(n);\ \ n := n + 1\\
\quad [\!]\ j \in r' \qquad\qquad\quad \rightarrow out!b(j);\ \ r' := r' \setminus \{j\};\ \ j := j + 1\\
\quad [\!]\ true \qquad\qquad\quad\ \rightarrow [k \geq j \rightarrow r' := r' \cup \{k\};\ \ b(k) := v [\!] k < j \rightarrow skip]\\
\quad [\!]\ \{0, ..n\} \cap s' \neq \emptyset \rightarrow k := any\{0, ..n\} \cap s';\ \ v := a(k)\\
\quad [\!]\ true \qquad\qquad\quad\ \rightarrow s' := i';\ \ l := \min(s')\\
\quad [\!]\ j \notin r' \qquad\qquad\quad \rightarrow i' := \{j, ..j + N\} \setminus r'\\
\quad]
\end{array}
$$

We may add probes on channels in and out to the guards of the first and second commands without affecting the program's correctness, since it was postulated that in and out communications succeed eventually. It might make the program more efficient, though.

$$
\begin{array}{l}
\overline{in} \wedge n \neq l + N \rightarrow in?a(n);\ \ n := n + 1\\
\overline{out} \wedge j \in r' \qquad \rightarrow out!b(j);\ \ r' := r' \setminus \{j\};\ \ j := j + 1
\end{array}
$$

5 Partition the program into processes

Next, we partition the variables and the actions into processes. The transformation that is used to go from the above sequential program to a distributed program is: transform every guarded command

$$ga \wedge gb \rightarrow x := e$$

into two guarded commands. Introduce a channel, c say, and choose either the pair

$$
\begin{array}{l}
ga \qquad\ \rightarrow c!e\\
gb \wedge \overline{c} \rightarrow c?x
\end{array}
$$

or the pair

$$
\begin{array}{l}
ga \wedge \overline{c} \rightarrow c!e\\
gb \qquad\ \rightarrow c?x
\end{array}
$$

The two guarded commands of a pair thus chosen are mapped to different processes. If the disjunction of the guards in a process is $true$, then the set GC of guarded commands is enclosed as $\ast[\ GC\ ]$, and otherwise as $\ast[true \rightarrow [\ GC\ ]]$. The latter variety corresponds to a nonterminating loop, the body of which is an if-statement that waits until at least one guard is $true$. (Of course, the first version can also be written in the second form without any change in effect.) If the interleaving of the processes is fair, if the selection between guarded commands in each process is fair, and if each process has either probes in all its guards or in none of its guards, then the processes implement the original program. (By "implement" we mean that it meets the safety and progress requirements of the original program.) In making the choices between the pairs, we make the choices such that we end up with the guarded commands that occur in the program text of the faulty channel. Here is the result. The channel from S to K is identified as channel sk, and similar for the other three channels. Channels ri and is transmit a set of integers per communication. Channels sk and kr transmit a pair: a message plus an integer (the sequence number of the message).

$$
\begin{array}{lllll}
K: & sk?(k,v); & & & \\
 & *[\ true \rightarrow sk?(k,v)\ [\!]\ true \rightarrow kr!(k,v)\ ] & & & \\
I: & ri?i'; & & & \\
 & *[\ true \rightarrow ri?i'\ [\!]\ true \rightarrow is!i'\ ] & & & \\
S: & l,n,s' := 0,1,\{0,..N\};\ in?a(0); & & & \\
 & *[true \rightarrow & [\ \overline{in} \wedge n \neq l+N & \rightarrow in?a(n);\ n := n+1 & \\
 & & [\!]\ \overline{sk} \wedge \{0,..n\} \cap s' \neq \emptyset & \rightarrow m:\ \{0,..n\} \cap s';\ sk!(m,a(m)) & \\
 & & [\!]\ \overline{is} & \rightarrow is?s';\ l := \min(s') & \\
 &] &] & & \\
R: & j,r' := 0,\emptyset; & & & \\
 & *[true \rightarrow & [\ \overline{out} \wedge j \in r' & \rightarrow out!b(j);\ r':\ r' \setminus \{j\};\ j := j+1 & \\
 & & [\!]\ \overline{kr} & \rightarrow kr?(h,u); & \\
 & & & \quad [h \geq j \rightarrow r' := r' \cup \{h\};\ b(h) := u [\!] h < j \rightarrow skip] & \\
 & & [\!]\ \overline{ri} \wedge j \notin r' & \rightarrow ri!\{j,..j+N\} \setminus r' & \\
 &] &] & &
\end{array}
$$

6 The alternating bit protocol

We conclude with a short section on the alternating bit protocol. This protocol has been studied and verified extensively in the literature. It is often pointed out that the sliding window protocol is a generalization and we support this claim by specializing our program to the case where $N=1$ and obtain the alternating bit protocol. We can slightly change the control structure of the program to take advantage of the fact $N=1$. For example, we now have that $n \neq l+N$ is *false* after an increase of n. We may therefore reduce the number of times that the test is performed by evaluating it after the update of l only. Doing the same thing at the receiver, we obtain the following program text for the alternating bit protocol. It uses the fact that sets i' and s' are singleton sets only. All sequence numbers are reduced modulo 2. The two channel processes do not change.

$$
\begin{array}{lll}
S: & n := 1;\ in?x; & \\
 & *[true \rightarrow [\ \overline{sk} \rightarrow sk!((n-1) \ \mathbf{mod}\ 2, x) & \\
 & \qquad\qquad [\!]\ \overline{is} \rightarrow is?l; & \\
 & \qquad\qquad\qquad [\ n = l \rightarrow in?x;\ n := (n+1)\ \mathbf{mod}\ 2\ [\!]\ n \neq l \rightarrow skip\ ] & \\
 & \quad] \qquad\quad] & \\
R: & j := 0; & \\
 & *[true \rightarrow [\ \overline{ri} \rightarrow ri!j & \\
 & \qquad\qquad [\!]\ \overline{kr} \rightarrow kr?(h, u); & \\
 & \qquad\qquad\qquad [\ h = j \rightarrow out!u;\ j := (j+1)\ \mathbf{mod}\ 2\ [\!]\ h \neq j \rightarrow skip\ ] & \\
 & \quad] \qquad\quad] &
\end{array}
$$

7 Concluding remarks

We have used channels that hold at most one message at a time. It is surprisingly easy to change the argument to the case where a channel may hold any number of messages, provided that the order of the messages is maintained. For example, if the return channel holds a sequence of messages this corresponds to a sequence of sets i. The essential property is that each one is a subset of the next element in the sequence. If the order is not maintained, this monotonicity property is lost. The program in which sequence numbers are not reduced modulo $2N$ can still be adapted to that situation (by replacing $s := i$ with $s := s \cup i$) but the version in which sequence numbers are reduced is beyond salvation.

In the alternating bit protocol, we simplified the sets of sequence numbers because it is was known that each sets contained at most one element. In the standard version of the sliding window protocol a similar simplification is made. For example, instead of set i, the number $\min(\overline{i})$ is transmitted (the "lowest missing number"). This change makes the program incorrect, however, because progress is no longer guaranteed. Here is a scenario that illustrates the problem. Assume $N \geq 2$. Suppose messages with sequence numbers l and $l+1$ have not yet been received by receiver R but they have been received by sender S ($n = l+2$). Instead of set $\{l, l+1\}$, integer l is sent from R via I to S. S sends both message l and message $l+1$. The faulty channel K looses the first message and delivers the second. As a result, the receiver misses only message l, and sends integer l via I to S. Again, S sends l and $l+1$, and the channel looses the first and delivers the second. The system has now reached the same state as before, the channels are fair (K looses every other message, I looses no messages) and the system is in a cycle without making progress. In our solution, which is known as the sliding window protocol with selective retry, set $\{l, l+1\}$ is sent from receiver R to sender S, but after message $l+1$ has successfully been received by r, set $\{l\}$ is transmitted. This causes sender S to send message l only instead of both l and $l+1$, which eventually causes l to be received by R.

A substantial number of papers report on safety properties of the sliding window protocols. Few papers address liveness issues. One of the problems with liveness properties is that they are not as readily formalized as safety properties are. Many specification languages do not include any form of temporal logic necessary for expressing them.

In [8] we find hardly any correctness considerations; only the issue of using cyclic numbers is addressed (and only by example). Of the references, however, it is the only one that mentions the selective retry that turned out to be essential for progress.

In [7] some safety properties are established (in an elegant way). We quote “however it is not shown that the protocol will progress”. In [6], the situation is similar. It establishes safety properties and also that “if the media are live then so too are the protocols”. However, this does not exclude livelock.

In [5] it is shown that the alternating bit protocol satisfies both safety and progress requirements. It is shown that the sliding window protocol (without selective retry) satisfies safety requirements and makes progress. The proofs are given for a stronger channel, however, viz. “if the same message is sent over and over again, it will eventually be delivered (provided that the receiving process repeatedly accepts messages)”. The difference with our weaker channels is that if repeatedly message A followed by message B is sent, then in our case it can only be guaranteed that every now and then a message arrives, possible only A’s and never a B. Hailpern’s stronger channels guarantee that both A and B arrive eventually. Although our scenario is unlikely if faulty behavior is random, it is not at all hard to construct a channel that looses every other message. Such a channel meets our requirements, but fails to satisfy Hailpern’s stronger requirements.

In [3] a formal verification of the sliding window protocol is presented. However, what is claimed to be a liveness property is actually a safety property. The protocol that is verified suffers from the lack of progress demonstrated by the above-mentioned scenario.

In [9] an interesting problem is discussed. Although some protocols are life they have the property that the number of steps it takes to deliver a message grows with the total number of communication faults that have occurred in the past. This is an undesirable property. It is the consequence of too strong a coupling between sender and receiver. (Namely, a receiver responds with a message to each message that is lost or received on the incoming channel.) In our case, the coupling is weak and the undesirable phenomenon does not occur.

The sequential programs that we used as intermediate steps in our transformation process are similar to Unity programs [4] and to Action systems [1]. The latter reference also discusses how these programs can be transformed into communicating sequential processes, similar to what we have done here.

Acknowledgement
I am very grateful to Peter Hofstee and Johan Lukkien for refusing to be satisfied with my earlier arguments and forcing me to do better, and to Ralph Back, Rustan Leino, and the members of IFIP WG 2.3 present at the Pouilly en Auxois meeting for discussions.

References

[1] R.-J.R. Back and R. Kurki-Suonio. Decentralization of process nets with centralized control. *Distributed Computing*, (3):73–87, 1989.

[2] K.A. Bartlett, R.A. Scantlebury, and P.T. Wilkinson. A note on reliable full-duplex transmission over half-duplex links. *Communications of the ACM*, 12(5):260–261, 1969.

[3] R. Cardell-Oliver. Using higher order logic for modelling real-time protocols. In *TAPSOFT '91*, volume 494 of *Lecture Notes in Computer Science*, pages 259–282. Springer-Verlag, 1991.

[4] K.M. Chandy and J. Misra. *Parallel Program Design, a foundation.* Addison-Wesley, 1988.

[5] B.T. Hailpern. *Verifying Concurrent Processes Using Temporal Logic*, volume 129 of *Lecture Notes in Computer Science*. Springer-Verlag, 1982.

[6] K. Paliwoda and J.W. Sanders. An incremental specification of the sliding-window protocol. *Distributed Computing*, 5:83–94, 1991.

[7] N. V. Stenning. A data transfer protocol. *Computer Networks*, 1(2):99–110, 1976.

[8] A.S. Tanenbaum. *Computer Networks, second edition.* Prentice Hall, 1988.

[9] Y. Yemini and J.F. Kurose. Can current protocol verification techniques guarantee correctness? *Computer Networks*, 6:377–381, 1982.