The Sliding-Window Protocol Revisited

Jan L.A. van de Snepscheut

Computer Science, California Institute of Technology, Pasadena, California, USA

Keywords: Communication protocols; Program transformation; Guarded commands; Fairness

Abstract. 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.

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 [BSW69]. We have been unable to trace back the origins of the sliding-window protocols; [Ste76] discusses one of the versions and lists networks using related protocols.

2. Program Notation

For the sequential part of the algorithms, we use a subset of Edsger W. Dijkstra's guarded command language [Dij76]. Sometimes we will assume the choice between alternatives in a guarded command to be weakly fair. By weakly fair we mean that no guarded command is ignored infinitely often, hence if a guard remains *true* the corresponding statement is executed eventually. We write $*[\langle i :: B_i \rightarrow S_i \rangle]$ for **do** $\langle i :: B_i \rightarrow S_i \rangle$ **od** and $*[S]$ for **do** *true* $\rightarrow S$ **od**. For

(sequential) statements $S0$ and $S1$, statement $S0\|S1$ denotes their concurrent execution. The (sequential) constituents $S0$ and $S1$ are then called processes. Execution of a parallel composition corresponds to a weakly fair interleaving of the atomic actions of the composed processes. The semantics of the **if** statement is reinterpreted (cf. [Hoa78]). If all guards of an **if** statement evaluate to *false*, its execution results in suspension rather than **abort**. When the guards contain no shared variables, an **if** statement that is suspended remains suspended forever and therefore this definition is compatible with the original semantics.

The semantics of communication primitives is as described in [Mar81]. The main difference with C.A.R. Hoare's proposal in [Hoa78] is in the naming of channels rather than processes. In [Hoa85], the same author proposes to name channels instead of processes in communication commands. We use channels in a similar way as variables, hence hiding is accomplished through the regular scoping mechanism. We summarize the axioms from [Mar81] for our channels. For an input or output action A on a channel a , let ca denote the number of completed A actions, i.e., the number of times that command A has been executed since initiation of the program's execution. The *synchronization requirement* for channel a is that $ca? = ca!$ holds at any point in the computation.

Note: It is sometimes attractive to weaken the synchronization requirement by putting some bound on $ca! - ca?$. This may be a lower bound only, or both a lower and an upper bound. The maximum difference between the two is then called the *slack*, since it indicates how far the synchronized processes can get out of step. The use of a nonzero slack sometimes leads to minor complications in proofs and definitions. **end of Note**

The execution of a command results either in the completion of the action or in its suspension when its completion would violate the synchronization requirement. From suspension until completion an action is *pending* and the process executing the action is delayed. Boolean valued qa is equal to the predicate "an A action is pending". The *progress requirement*, $\neg qa! \vee \neg qa?$, states that actions are suspended only if their completion would violate the synchronization requirement. The n th $a!$ action is said to *match* the n th $a?$ action. The completion of a matching pair of communication actions is called a *communication*. The *communication requirement*, the third and final axiom, states that execution of matching actions $a!E$ and $a?v$ amounts to the assignment $v := E$. We shall also use probes [Mar85] on channels. Probe a , denoted by $\bar{a}$, is *true* just when a process is suspended on a communication on a .

3. 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 such a 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 which arrive do so 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 each make a fair (but not necessarily random) choice between *true* and *false*; in any sequence of calls only a finite number of either value occurs consecutively. Channels c and d are fault free.

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

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.

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 assumption is valid since 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.

```

var d : channel;
(
  *[c?x; * [flip \rightarrow d!(x, flip')]]
  ||
  *[d?(y, b); [b \rightarrow skip || \neg b \rightarrow e!y]]
)

```

The latter combination can be replaced by

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

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 [ChM88].

4. The Sliding-Window Protocol

In this section we consider the sliding-window protocol. We have a network as indicated in the figure below. Channels F (forward) and B (backward) are faulty, and the task is to construct programs S (send) and R (receive) 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 and subsequently refine the program to contain only bounded sequence numbers. The alternative of starting with the latter program would require the introduction of the unbounded integers for proving the program's correctness anyway.

We seek a program that guarantees that the sequence of values transmitted via out is a prefix of the sequence of values transmitted via in . Their 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 F and B . The sequential program uses variables s and r whose values equal the number of completed input and output operations respectively. The essential invariant is

$$s = \#in \wedge r = \#out \wedge r \leq s \leq r + N \wedge \\ \langle \forall h : 0 \leq h < s : a(h) = in_h \rangle \wedge \langle \forall h : 0 \leq h < r : a(h) = out_h \rangle$$

in which in_h (out_h) is the h th value communicated via channel in (out), and $a(i : 0 \leq i)$ is an array of (for now) unbounded size. The program performs two actions

$$in?a(s); s := s + 1 \\ out!a(r); r := r + 1$$

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

$$s, r := 0, 0; \\ * [s \neq r + N \wedge \overline{in} \rightarrow in?a(s); s := s + 1 \\ \parallel r \neq s \wedge \overline{out} \rightarrow out!a(r); r := r + 1 \\]$$

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. With this in mind, we have chosen names for the variables such that the first letter refers to the process of which the variable is to become a local variable. A requirement is that each action operate on the local variables of one process only. For example, variable s , which stores the number of *in?* operations, will be assigned to process S because channel *in* is connected to S . Similarly, r is assigned to R . As a result, no process stores both s and r , and therefore the guards $s \neq r + N$ and $r \neq s$ 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(s)* and *out!a(r)* operations for a while, though retaining variables s and r with the same interpretation. 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 to be used than would have been the case with perfectly reliable channels.

We use the notation $[i..j)$ to denote a set of integers defined by

$$x \in [i..j) \equiv i \leq x < j$$

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 *rset*. The variable is going to be local to R and serves to eliminate variable s from the guard of action $r := r + 1$. The interpretation is

rset 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. Since r 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 s , we have invariant

$$[0..r) \subseteq rset \subseteq [0..s)$$

We need to add a guarded command to update *rset*. Defining

f is the number of the last message sent by S

we find that we need to add

$$rset := rset \cup \{f\}$$

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

$$0 \leq f < s$$

We need to add a command that assigns a new value to f . It is tempting to assign to f a value from the set $[0..s) - rset$ 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 s) and R (namely $rset$) and can therefore not be partitioned. We introduce variable $sset$, which is meant to be a copy of $rset$ but it might not be up to date. The latter is expressed by

$$sset \subseteq rset$$

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

$$f := [0..s) - sset$$

which assigns to f an arbitrary element of $[0..s) - sset$. 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 $sset$. It would be nice to write $sset := rset$ but this is not feasible because the communication between S and R is limited to communication via faulty channels. Therefore, we introduce intermediate variable $bset$ whose value is a copy of $rset$, but it might not be up to date. We say “intermediate” because its value will be somewhere in between $sset$ and $rset$. ($sset$ is a possibly-out-of-date copy of $bset$, and $bset$ is a possibly-out-of-date copy of $rset$.)

$$sset \subseteq bset \subseteq rset$$

This is the slightly stronger invariant referred to above. The commands for updating $sset$ and $bset$ are immediate.

$$sset := bset \quad \text{and} \quad bset := rset$$

It remains to introduce a local variable of S that tracks r so that we may update s . We may introduce a variable that is a copy of r , but not necessarily up to date. However, we already have such an approximation of r in the form of the smallest number missing from set $sset$. We introduce $\widetilde{set}$ as an abbreviation for $[0..\infty) - set$ and introduce variable sr .

$$sr = \min(\widetilde{sset})$$

Collecting all the terms of the invariant, we have

$$P : sr = \min(\widetilde{sset}) \leq \min(\widetilde{bset}) \leq r \leq s \leq sr + N \quad \wedge \quad 0 \leq f < s$$

$$\wedge \quad sset \subseteq bset \subseteq rset \subseteq [0..s) \quad \wedge \quad [0..r) \subseteq rset$$

in which the term $\min(\widetilde{sset}) \leq \min(\widetilde{bset})$ is redundant because it follows from $sset \subseteq bset$. The program developed thus far is

```

 $r, f, sr, s, sset, bset, rset := 0, 0, 0, 1, \emptyset, \emptyset, \emptyset;$ 
*[ ?  $\rightarrow s := s + 1$ 
  || ?  $\rightarrow r := r + 1$ 
  || ?  $\rightarrow rset := rset \cup \{f\}$ 
  || ?  $\rightarrow f := [0..s) - sset$ 
  || ?  $\rightarrow sset := bset; sr := \min(\widetilde{sset})$ 
  || ?  $\rightarrow bset := rset$ 
]
```

In order to enable initialization, the program starts with $s = 1$, which corresponds to performing an initial input action prior to executing the loop. We now calculate the guards of the six commands. The proof format we use is explained and motivated in [DijS90]. For predicate Q , Q_E^x stands for the predicate obtained from Q by textually replacing each occurrence of x in Q by E .

The first command is $s := s + 1$, and

$$\begin{aligned}
& P_{s+1}^s \\
= & \\
& sr = \min(\widetilde{sset}) \leq \min(\widetilde{bset}) \leq r \leq s + 1 \leq sr + N \quad \wedge \quad 0 \leq f < s + 1 \quad \wedge \\
& sset \subseteq bset \subseteq rset \subseteq [0..s + 1) \quad \wedge \quad [0..r) \subseteq rset \\
\Leftarrow & \\
& P \quad \wedge \quad s \neq sr + N
\end{aligned}$$

suggests guard $s \neq sr + N$.

The second command is $r := r + 1$, and

$$\begin{aligned}
& P_{r+1}^r \\
= & \\
& sr = \min(\widetilde{sset}) \leq \min(\widetilde{bset}) \leq r + 1 \leq s \leq sr + N \quad \wedge \quad 0 \leq f < s \quad \wedge \\
& sset \subseteq bset \subseteq rset \subseteq [0..s) \quad \wedge \quad [0..r + 1) \subseteq rset \\
\Leftarrow & \\
& P \quad \wedge \quad r < s \quad \wedge \quad r \in rset \\
= & \quad \{ P \Rightarrow rset \subseteq [0..s) \} \\
& P \quad \wedge \quad r \in rset
\end{aligned}$$

suggests guard $r \in rset$.

The third command is $rset := rset \cup \{f\}$, and

$$\begin{aligned}
& P_{rset \cup \{f\}}^{rset} \\
= & \\
& sr = \min(\widetilde{sset}) \leq \min(\widetilde{bset}) \leq r \leq s \leq sr + N \quad \wedge \quad 0 \leq f < s \quad \wedge \\
& sset \subseteq bset \subseteq rset \cup \{f\} \subseteq [0..s) \quad \wedge \quad [0..r) \subseteq rset \cup \{f\} \\
\Leftarrow & \\
& P
\end{aligned}$$

suggests guard *true*.

The fourth command is $f := [0..s) - sset$, and

$$\begin{aligned}
& wp.P.(f := [0..s) - sset) \\
= & \{ wp.P.(f := T) = (T \neq \emptyset) \quad \wedge \quad \langle \forall t : t \in T : P_t^f \rangle \} \\
& [0..s) - sset \neq \emptyset \quad \wedge \quad sr = \min(\widetilde{sset}) \leq \min(\widetilde{bset}) \leq r \leq s \leq sr + N \quad \wedge \\
& \langle \forall g : g \in [0..s) - sset : 0 \leq g < s \rangle \quad \wedge
\end{aligned}$$

$$\begin{aligned}
& sset \subseteq bset \subseteq rset \subseteq [0..s) \wedge [0..r) \subseteq rset \\
\Leftarrow \\
& P \wedge [0..s) - sset \neq \emptyset
\end{aligned}$$

suggests guard $[0..s) - sset \neq \emptyset$.

The fifth command is $sset := bset; sr := \min(\widetilde{sset})$, and

$$\begin{aligned}
& (P_{\min(\widetilde{sset})}^{sr})_{bset}^{sset} \\
= & \\
& \min(\widetilde{bset}) = \min(\widetilde{bset}) \leq \min(\widetilde{bset}) \leq r \leq s \leq \min(\widetilde{bset}) + N \wedge \\
& 0 \leq f < s \wedge bset \subseteq bset \subseteq rset \subseteq [0..s) \wedge [0..r) \subseteq rset \\
\Leftarrow \\
& P
\end{aligned}$$

suggests guard *true*.

The sixth command is $bset := rset$, and

$$\begin{aligned}
& P_{rset}^{bset} \\
= & \\
& sr = \min(\widetilde{sset}) \leq \min(\widetilde{rset}) \leq r \leq s \leq sr + N \wedge 0 \leq f < s \wedge \\
& sset \subseteq rset \subseteq rset \subseteq [0..s) \wedge [0..r) \subseteq rset \\
\Leftarrow \\
& P \wedge \min(\widetilde{rset}) \leq r \\
\Leftarrow \\
& P \wedge r \notin rset
\end{aligned}$$

suggests guard $r \notin rset$. Thus, we obtain the complete program.

$$\begin{aligned}
& r, f, sr, s, sset, bset, rset := 0, 0, 0, 1, \emptyset, \emptyset, \emptyset; \\
& * [s \neq sr + N \quad \rightarrow s := s + 1 \\
& \quad \parallel r \in rset \quad \rightarrow r := r + 1 \\
& \quad \parallel true \quad \rightarrow rset := rset \cup \{f\} \\
& \quad \parallel [0..s) - sset \neq \emptyset \rightarrow f := [0..s) - sset \\
& \quad \parallel true \quad \rightarrow sset := bset; sr := \min(\widetilde{sset}) \\
& \quad \parallel r \notin rset \quad \rightarrow bset := rset \\
&]
\end{aligned}$$

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

$$(r + \#sset + \#bset + \#rset + s, sset \neq rset \vee f \notin rset)$$

The first component of the pair is an integer, the second component is a boolean. The ordering of the pair is the 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 we can partition the state space in a way that meets the following two conditions. First, in each partition a guarded command exists whose guard reduces to *true* for all states in that partition and whose body strictly increases the variant function. Second, no other command causes the program to leave the partition without strictly increasing the variant function. If we postulate a weak, fair choice between the guarded commands of the loop, progress follows.

- If $sset \neq bset$, guarded command $true \rightarrow sset := bset$ increases the variant function; the condition is not falsified by any other command. (The last command never falsifies the condition due to the invariance of $sset \subseteq bset \subseteq rset \subseteq [0..s]$.)
- If $r \in rset$, guarded command $r \in rset \rightarrow r := r + 1$ increases the variant function; the condition is not falsified by any other command.
- If $bset \neq rset \wedge r \notin rset$, guarded command $r \notin rset \rightarrow bset := rset$ increases the variant function; the condition can be falsified by only one other command, viz., $rset := rset \cup \{f\}$ if $r = f \wedge f \notin rset$, which also increases the variant function.
- If $sset = bset = rset \neq [0..s] \wedge f \in rset$, guarded command $[0..s] - sset \neq \emptyset \rightarrow f \in [0..s] - sset$ increases the variant function; the condition is not falsified by any other command. (The first command never falsifies the condition due to the invariance of $sset \subseteq bset \subseteq rset \subseteq [0..s]$.)
- If $sset = bset = rset \neq [0..s] \wedge f \notin rset$, guarded command $true \rightarrow rset := rset \cup \{f\}$ increases the variant function; the condition is not falsified by any other command. (The first command never falsifies the condition due to the invariance of $sset \subseteq bset \subseteq rset \subseteq [0..s]$. The fourth command may assign a new value to f which is not in $sset$ and therefore, given the condition, not in $rset$. Hence this command also does not change the condition.)
- If $sset = bset = rset = [0..s]$, guarded command $s \neq sr + N \rightarrow s := s + 1$ increases the variant function; its guard is *true* (since $N > 0$). If $f \notin rset$ the condition may be falsified by $rset := rset \cup \{f\}$ which under that condition increases the variant function.

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

5. The Range of f and r

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

$$\begin{aligned}
 r, f, sr, s, sset, bset, rset, fset &:= 0, 0, 0, 1, \emptyset, \emptyset, \emptyset, \emptyset; \\
 * [s \neq sr + N &\rightarrow s := s + 1 \\
 \parallel r \in rset &\rightarrow r := r + 1 \\
 \parallel true &\rightarrow rset := rset \cup \{f\}
 \end{aligned}$$

```


$$\begin{aligned}
& \parallel [0..s) - sset \neq \emptyset \rightarrow f := [0..s) - sset; fset := sset \\
& \parallel true \quad \rightarrow sset := bset; sr := \min(\widetilde{sset}) \\
& \parallel r \notin rset \quad \rightarrow bset := rset \\
& ]
\end{aligned}$$


```

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

$$f \in [0..s) - fset \quad \wedge \quad f < \min(\widetilde{fset}) + N \quad \wedge \quad r \leq \min(\widetilde{fset}) + N \quad \wedge \\ fset \subseteq sset \quad \wedge \quad rset \subseteq [0.. \min(\widetilde{fset}) + N)$$

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 $fset := sset$ does not decrease $\min(\widetilde{fset})$ since we have $fset \subseteq sset$. From the new and the old invariants, we have

$$\min(\widetilde{fset}) \leq r \leq \min(\widetilde{fset}) + N \quad \wedge \quad \min(\widetilde{fset}) \leq f < \min(\widetilde{fset}) + N$$

(observe $f \in \widetilde{fset}$) from which we conclude

$$-N < r - f \leq +N$$

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

$$sr \leq s \leq sr + N$$

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 $rset$, we may introduce $rset'$ such that

$$rset' = rset - [0..r)$$

(because the invariant implies $[0..r) \subseteq rset$). Also, because the invariant implies $\min(\widetilde{sset}) \leq s \leq \min(\widetilde{sset}) + N$, we may introduce

$$sset' = \widetilde{sset} \cap [\min(\widetilde{sset}).. \min(\widetilde{sset}) + N)$$

and similarly

$$bset' = \widetilde{bset} \cap [\min(\widetilde{bset}).. \min(\widetilde{bset}) + N)$$

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 arrays of messages a and b and a message variable v ; their use is governed by invariant

$$\langle \forall h : 0 \leq h < s : a(h) = in_h \rangle \quad \wedge \quad \langle \forall h : 0 \leq h < r : out_h = in_h \rangle \quad \wedge \\ \langle \forall h : h \in rset : b(h) = in_h \rangle \quad \wedge \quad v = in_f$$

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

$$\begin{aligned}
& r, sr, s, sset', rset' := 0, 0, 1, [0..N), \emptyset; \\
& in?a(0); f, v := 0, a(0); bset' := sset'; \\
& *[s \neq sr + N \rightarrow in?a(s); s := s + 1 \\
& \quad \parallel r \in rset' \rightarrow out!b(r); rset' := rset' - \{r\}; r := r + 1 \\
& \quad \parallel true \rightarrow [f \geq r \rightarrow rset' := rset' \cup \{f\}; b(f) := v \\
& \quad \quad \parallel f < r \rightarrow skip \\
& \quad] \\
& \parallel [0..s) \cap sset' \neq \emptyset \rightarrow f := [0..s) \cap sset'; v := a(f) \\
& \parallel true \rightarrow sset' := bset'; sr := \min(sset') \\
& \parallel r \notin rset' \rightarrow bset' := [r..r + N) - rset' \\
&]
\end{aligned}$$

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.

$$\begin{aligned}
& \overline{in} \wedge s \neq sr + N \rightarrow in?a(s); s := s + 1 \\
& \overline{out} \wedge r \in rset' \rightarrow out!b(r); rset' := rset' - \{r\}; r := r + 1
\end{aligned}$$

6. 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

$$ga \rightarrow c!e \quad gb \wedge \bar{c} \rightarrow c?x$$

or the pair

$$ga \wedge \bar{c} \rightarrow c!e \quad gb \rightarrow c?x$$

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 $*[GC]$, and otherwise as $*[true \rightarrow [GC]]$. 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 probes either 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 F is identified as channel sf , and similarly for the other three channels. Channels rb and bs transmit a set of integers per communication. Channels sf and fr transmit a pair: a message plus an integer (the sequence number of the message).

```

F :  sf?(f,v);
    * [ true → sf?(f,v) || true → fr!(f,v) ]

B :  rb?bset';
    * [ true → rb?bset' || true → bs!bset' ]

S :  sr,s,sset' := 0,1,[0..N); in?a(0);
    * [ true → [  $\overline{in} \wedge s \neq sr + N$            → in?a(s); s := s + 1
                  ||  $\overline{sf} \wedge [0..s) \cap sset' \neq \emptyset \rightarrow m : \in [0..s) \cap sset'; sf!(m,a(m))$ 
                  ||  $\overline{bs}$                                → bs?sset'; sr := min(sset')
                ]
    ]

R :  r,rset' := 0,∅;
    * [ true → [  $\overline{out} \wedge r \in rset' \rightarrow out!b(r); rset' := rset' - \{r\};$ 
                  r := r + 1
                  ||  $\overline{fr}$                                → fr?(h,u);
                  [ h ≥ r → rset' := rset' ∪ {h}; b(h) := u
                  || h < r → skip
                  ]
                  ||  $\overline{rb} \wedge r \notin rset' \rightarrow rb!([r..r + N) - rset'$ 
    ]

```

7. 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 change the control structure of the program slightly to take advantage of the fact $N = 1$. For example, we now have that $s \neq sr + N$ is *false* after an increase of s . We may therefore reduce the number of times that the test is performed by evaluating it after the update of sr 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 $bset'$ and $sset'$ are singleton sets only.

All sequence numbers are reduced modulo 2. The two channel processes do not change.

$$\begin{aligned}
 S : \quad & s := 1; \text{ in } ?x; \\
 & * [\text{ true } \rightarrow [\overline{sf} \rightarrow sf !((s-1) \bmod 2, x) \\
 & \quad \parallel \overline{bs} \rightarrow bs ?sr; \\
 & \quad [s = sr \rightarrow \text{ in } ?x; s := (s+1) \bmod 2 \\
 & \quad \parallel s \neq sr \rightarrow \text{ skip} \\
 &] \quad] \quad] \\
 R : \quad & r := 0; \\
 & * [\text{ true } \rightarrow [\overline{rb} \rightarrow rb !r \\
 & \quad \parallel \overline{fr} \rightarrow fr ?(h, u); \\
 & \quad [h = r \rightarrow \text{ out } !u; r := (r+1) \bmod 2 \\
 & \quad \parallel h \neq r \rightarrow \text{ skip} \\
 &] \quad] \quad]
 \end{aligned}$$

8. 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 $bset$. 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 $sset := bset$ with $sset := sset \cup bset$) 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 was known that each set contained at most one element. In the standard version of the sliding-window protocol a similar simplification is made. For example, instead of set $bset$, the number $\min(bset)$ 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 sr and $sr+1$ have not yet been received by receiver R but they have been received by sender S ($s = sr+2$). Instead of set $\{sr, sr+1\}$, integer sr is sent from R via B to S . S sends both message sr and message $sr+1$. The faulty channel K loses the first message and delivers the second. As a result, the receiver misses only message sr , and sends integer sr via B to S . Again, S sends sr and $sr+1$, and the channel loses the first and delivers the second. The system has now reached the same state as before, the channels are fair (F loses every other message, B loses 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 $\{sr, sr+1\}$ is sent from receiver R to sender S , but after message $sr+1$ has successfully been

received by R , set $\{sr\}$ is transmitted. This causes sender S to send message sr only instead of both sr and $sr + 1$, which eventually causes sr 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 [Tan88] we find almost no 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 [Ste76] some safety properties are established (in an elegant way). We quote “however it is not shown that the protocol will progress”. In [PaS91], 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 [Hai82] 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 message A followed by message B is sent repeatedly, then in our case it can only be guaranteed that every now and then a message arrives, possibly 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, a channel that loses every other message meets our requirements, but fails to satisfy Hailpern's stronger requirements.

In [COI91] 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 [YeK82] an interesting problem is discussed. Although some protocols are live 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 [ChM88] and to Action systems [BKS89]. The latter reference also discusses how these programs can be transformed into communicating sequential processes, in the way 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

- [BKS89] Back, R.-J. R. and Kurki-Suonio, R.: Decentralization of process nets with centralized control. *Distributed Computing*, (3):73–87, 1989.
- [BSW69] Bartlett, K. A., Scantlebury, R. A. and Wilkinson, P. T.: A note on reliable full-duplex transmission over half-duplex links. *Communications of the ACM*, 12(5):260–261, 1969.
- [ChM88] Chandy, K. M. and Misra, J.: *Parallel Program Design, a foundation*. Addison-Wesley, 1988.
- [COI91] Cardell-Oliver, R.: Using higher order logic for modelling real-time protocols. In *TAP-SOFT '91*, volume 494 of *Lecture Notes in Computer Science*, pages 259–282. Springer-Verlag, 1991.
- [Dij76] Dijkstra, E. W.: *A Discipline of Programming*. Prentice-Hall, 1976.
- [DijS90] Dijkstra, E. W. and Scholten, C. S.: *Predicate Calculus and Program Semantics*. Springer-Verlag, 1990.
- [Hai82] Hailpern, B. T.: *Verifying Concurrent Processes Using Temporal Logic*, volume 129 of *Lecture Notes in Computer Science*. Springer-Verlag, 1982.
- [Hoa78] Hoare, C. A. R.: Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.
- [Hoa85] Hoare, C. A. R.: *Communicating Sequential Processes*. Series in Computer Science (C.A.R. Hoare, ed.). Prentice-Hall International, 1985.
- [Mar81] Martin, A. J.: An axiomatic definition of synchronization primitives. *Acta Informatica*, 16:219–235, 1981.
- [Mar85] Martin, A. J.: The probe: an addition to communication primitives. *Information Processing Letters*, 20:125–130, 1985. and (21)107.
- [PaS91] Paliwoda, K. and Sanders, J. W.: An incremental specification of the sliding-window protocol. *Distributed Computing*, 5:83–94, 1991.
- [Ste76] Stenning, N. V.: A data transfer protocol. *Computer Networks*, 1(2):99–110, 1976.
- [Tan88] Tanenbaum, A. S.: *Computer Networks, second edition*. Prentice Hall, 1988.
- [YeK82] Yemini, Y. and Kurose, J. F.: Can current protocol verification techniques guarantee correctness? *Computer Networks*, 6:377–381, 1982.

Submitted to Distributed Computing in 1992