A Formal Derivation of a Sliding Window Protocol

Rob R. Hoogerwoord

Contents

Chapter 0

Introduction

0.0 On systematic protocol design

My main point of view is that a distributed algorithm just is a parallel algorithm, with the additional property that its variables can be partitioned in such a way that their values can be stored, without too much trouble, in the machines on which the distributed algorithm will be implemented. In particular, it is desirable that all modifications of a variable are confined to components that are destined to be excuted by one and the same machine in the distributed system.

In addition, inspections of the values of shared variables require some care. If the value of a variable is needed outside the machine holding its value, some form of communication must be used to send that value to where it is needed. That is, in a truly distributed system, shared variables are neither modified nor inspected outside the machines holding their values, and all interaction between components in different machines take place via communication.

A viable design strategy now is to view designing a distributed algorithm as designing an ordinary parallel algorithm, without too much regard, at least initially, for the requirement of distribution. In a way, the latter can (and should) be viewed as an implementation issue, to be dealt with at the right moment, which usually means: not too early in the design. In particular, very often the program can and, therefore should, be conveniently formulated in terms of (a limited use of) shared variables. Communication by means of (synchronous or asynchronous) transmission of items is a way to implement these shared variables in a distributed way; preferably, this is introduced into the game only after the main design is complete, almost as an afterthought, so to speak. The advantage of this approach is that the design evolves in a need-driven way: only when the main design has been completed it becomes clear what form and amount of communication is needed.

In earlier studies, like [3] and [4], I have demonstrated the feasibility of this approach. Here we illustrate the approach with yet another example, namely the systematic construction of a (so-called) Sliding Window Protocol. This problem distinguishes itself from the earlier studies in two respects:

- several aspects can be distinguished that admit a clear Separation of Concerns;
- the design is largely driven by progress considerations.

0.1 Sliding Window Protocols

There is no such thing as "the" Sliding Window Protocol. Actually, the name refers to a whole *class* of protocols that differ in their details.

A Sliding Window Protocol provides an efficient solution to the problem how to transfer a sequence of *(data) items* – a (potentially infinite) data stream – via an *unreliable* communication channel. The communication channel is unreliable in the following ways:

- Any item sent through the channel needs not arrive at the receiving end; that is, items sent may become *lost*. To be able to guarantee progress we have to assume, however, that not all items are lost: if sufficiently many items have been sent sufficiently many items also will arrive; otherwise, the channel would be completely useless.
- Any item sent through the channel may arrive more than once at the receiving end; that is, items may be *duplicated* by the channel. Again, to prevent the channel from becoming completely useless, we have to assume that the amount of duplication is finite.
- The channel does not (necessarily) have the FIFO-property; that is, items need not arrive at the receiving end in the same order in which they have been sent.

Despite the channel's unreliability, the protocol has to reliably provide transmission of the items offered by the sending side – from now onwards called "the Sender" – to the receiving end – from now onwards called "the Receiver" –. This transmission is required to have the FIFO-property: all items are to be delivered to the Receiver in the same order in which the Sender has offered them for transmission.

For the purpose of the protocol a second (unreliable) communication channel is available, from the Receiver back to the Sender, via which, for example, acknowledgements may be sent back. To distinguish the two communication channels, we will call the channel from the Sender to the Receiver the *forward (communication)*

channel and we will call the other channel the *backward (communication) channel*. Both channels are unreliable as defined above.

As far as I know, the oldest discussion of such a Sliding-Window Protocol is a publication [8] by Stenning, from 1976. Stenning's presentation is remarkably modern, in that he formulates invariants and attempts to give a proof of the correctness of the algorithm. The only other formal treatment of the subject I know of is a (post-mortem) publication [7] by van de Snepscheut. His approach is to start with a sequential program and to gradually transform it, preserving its correctness, into a distributed program. Van de Snepscheut uses synchronous communication and assumes the communication channels to have the FIFO-property; like ours, van de Snepscheut's treatment also can be considered as a systematic attempt to *design* the protocol.

0.2 On the method

We use a method for development of parallel programs that is based on the theory by Owicki and Gries, as originally presented in [6] and as further developed into a design discipline by Feijen and van Gasteren in [2].

The main ingredient of this discipline is the notion of *global correctness*, which entails that every assertion in every parallel *component* of a parallel program must be *globally correct*. This means that every such assertion is invariant under – that is, not violated by – every atomic statement in every other component of the parallel program.

Techniques have been developed to keep the number of global proof obligations manageable and to use these proof obligations as guiding principles for program construction. We do not summarize these techniques here, as we assume the reader to be familiar with [2].

A technique that is of particular importance and that deserves to be mentioned separately is that it is possible to construct the annotation – that is, the whole of all assertions – in the program in a step-by-step fashion: both *adding* a new assertion and *strengthening* an already present assertion never violates the correctness of any assertions already present in the program: all we have to do is to prove (local and global) correctness of the new assertion.

In the same vein, *guards* both in selections and in synchronizing statements may be safely strengthened without violating the correctness of the existing annotation; we refer to [2] for details.

0.2.0 Establish statements

For purposes of synschronization we use, so-called, *establish* statements, with this syntactic shape, in which B is a boolean expression:

$$\mathsf{est}\ B\ .$$

This statement is only eligible for execution in states where expression B has value true, but its execution entails no state change whatsoever: it is a guarded skip. In states where the value of B is false the statement is blocking the component in which it occurs: then the component's execution is postponed until the expression has become true.

The proof rule for (local) correctness of postcondition Q in:

$$\{\ P\ \}\ \mathsf{est}\ B\ \{\ Q\ \}\ ,$$

is:

$$[\ P \wedge B\ \Rightarrow\ Q\ ]\ \text{, which can also be written (equivalently) as:}$$
$$[\ P\ \Rightarrow\ (B \Rightarrow Q)\ ]\ \text{, or as:}$$
$$[\ B\ \Rightarrow\ (P \Rightarrow Q)\ ]\ .$$

0.2.1 Selection statements

In addition we use, so-called, *selection* statements, with this syntactic shape, in which k is variable and B is a boolean expression – "k such that B" –:

$$k : B\ .$$

This statement is equivalent to the assignment to variable k of any – not further specified – value for which boolean expression B has value true. If no such value exists at the time of evaluation of B the statement is blocking the component in which it occurs: then the component's execution is postponed until the expression has become true for at least one possible value of k. Thus, selection statements can be used for synchronisation. Actually, we can think of the selection proper as having been prefixed with a synchronizing statement "$\mathsf{est}\ (\exists k :: B)$".

For example, $k : sp \le k < sq$ assigns a value to k in the interval $[\,sp\,..\,sq\,)$; if $sp < sq$ such a value exists and the statement's execution terminates, but as long as $sp \ge sq$ the interval is empty and the statement is blocking.

0.3 Unreliable communication channels

0.3.0 The rule of Import and Export

As we are heading for the design of a distributed communication protocol, we need communication primitives that, in effect, constitute the interface between the protocol and the unreliable communication channels. In the final protocol, the above mentioned synchronization statements will only be used for *local* purposes, that is, for synchronization between parallel components within a single machine, either the Sender or the Receiver.

No matter how unreliable the communication channels are, in one respect they are extremely predictable: they do not make up items as every item delivered to the receiving end has been sent once from the sending end. This is reflected in the following Rule of Import and Export.

Communication is *asynchronous*, which means that sending and receiving an item are two distinct events that do not – as in the case of synchronous communication – form a single indivisible action.

Sending an item E is denoted as a send-statement – with pre- and postcondition P –:

$$\{\ P\ \}\ \mathsf{send}\ E\ \{\ P\ \}\quad ,$$

whereas receiving an item into a local variable x is denoted as a receive-statement – with pre- and postconditions Q and R –:

$$\{\ Q\ \}\ \mathsf{receive}\ x\ \{\ R\ \}\quad .$$

The Rule of Import and Export then states that local correctness of R amounts to:

$$[\ P \wedge Q\ \Rightarrow\ R(x := E)\ ]\quad .$$

In this study we will apply this rule tacitly, without explicitly referring to it: whenever we have introduced a receive-statement with a desired postcondition R we will see to it that, as far as R's local correctness does not follow from the statements precondition, every send-statement has $R(x := E)$ as its precondition.

The Rule of Import and Export is only valid for communication via the *same* communication channel. In Sliding Window Protocols only two communication channels play a role, which we have dubbed the "forward channel" and the "backward channel". To distinguish communication via these channels we use primitives send and receive for forward communication and we use send-ack and receive-ack for backward communication.

Receive-statements are (potentially) blocking: an item cannot be received before it has been sent. Send-statements are assumed to be non-blocking and are assumed to be always executable. Depending on the actual communication channels this assumption may or may not be realistic.

0.3.1 How to keep unbounded nondeterminism manageable

The prototype of a (sequential) program with unbounded nondeterminism has the following shape, in which X and Y are natural constants:

$$
\begin{array}{l}
\quad \{\, 0 \le X \,\wedge\, 0 \le Y \,\} \\
\quad x, y := X, Y \\
;\ \{\, \mathsf{invariant}\colon\ 0 \le x \,\wedge\, 0 \le y \,\} \\
\quad \mathsf{do}\ \ 0 \ne x \ \rightarrow\ \ x := x-1 \ ;\ y := \text{“any natural number”} \\
\quad [\!]\ \ 0 \ne y \ \rightarrow\ \ y := y-1 \\
\quad \mathsf{od}
\end{array}
$$

Here the statement $y :=$ “any natural number” indicates that, each time this assignment is executed, variable y is given an arbitrary natural number as its new value that is not a priori bounded: any natural number is as good as any other.

An obvious invariant of the repetition is $0 \le x \,\wedge\, 0 \le y$. Although execution of the repetition may require a number of steps that is unknown in advance, termination is guaranteed nevertheless: the pair $\langle\, x, y \,\rangle$, with lexicographical ordering, serves as variant function, as we observe that both $\langle\, x-1, n \,\rangle$, for any natural number n, and $\langle\, x, y-1 \,\rangle$ are lexicographically less than $\langle\, x, y \,\rangle$. In addition, the lexicographical order on pairs of naturals is well-founded; thus, termination of the repetition is guaranteed.

Yet, we can represent this form of unbounded nondeterminism in a deterministic way, by means of *clairvoyance*: we assume – for the sake of argument only – that the successive natural numbers to be assigned to y in the repetition's body are available in an infinite array $B[\,0\,..\,\infty\,)$, say, which is assumed available in advance. For each execution of $y :=$ “any natural number” the first “unused element” from array B is taken; this is implemented by means of an additional index variable p, say, as follows:

$$
\begin{array}{l}
\quad \{\, 0 \le X \,\wedge\, 0 \le Y \,\} \\
\quad x, y, p := X, Y, 0 \\
;\ \{\, \mathsf{invariant}\colon\ 0 \le x \,\wedge\, 0 \le y \,\wedge\, 0 \le p \,\} \\
\quad \mathsf{do}\ \ 0 \ne x \ \rightarrow\ \ x := x-1 \ ;\ y := B_p \ ;\ p := p+1 \\
\quad [\!]\ \ 0 \ne y \ \rightarrow\ \ y := y-1 \\
\quad \mathsf{od}
\end{array}
$$

As each increase of p is accompanied by a decrease of x, an additional invariant of the repetition is that $x+p$ is constant; as the initial value of $x+p$ is X, so is this constant, and the value of p is bounded accordingly: $p \leq X$ is invariant too.

Having thus eliminated the unbounded nondeterminism, termination of the repetition can now be proved simply by means of a natural-valued variant function, whose initial value is an upper bound for the number of steps the repetition will take. This variant function is:

$$x + y + (\Sigma i : p \leq i < X : B_i) \quad ,$$

the initial value of which is $X + Y + (\Sigma i : 0 \leq i < X : B_i)$. Notice that this equals $X + Y$ plus the (finite) sum of all values assigned to y during the repetition.

0.3.2 How to model a duplicating channel

In a very similar way, a channel that is allowed to duplicate items arbitrary but finite amounts of times can be modelled by assigning an arbitrary natural number to each item. This number then represents the number of (identical) copies of the item that will be delivered at the receiving end of the channel. Just as in the previous subsection, instead of using unbounded nondeterminism, in phrases like "any natural number", we use an infinite array $D[0..\infty)$ from which these numbers will be drawn, together with an additional index variable representing the number of items sent through the channel. Array D is constant, with the interpretation that D_n is the maximal number of copies of item n, where n is the ordinal number of the item, counted from 0 onwards; that is, an item sent gets ordinal number n whenever n items already have been sent before it.

We assume all elements of D to be *positive*: $D_n = 1$ represents the case where item n is not duplicated and will be delivered only once, whereas $D_n > 1$ represents the case where item n is duplicated $D_n - 1$ times.

To record how many copies of an item are actually "still under way" we will use an auxiliary variable $d[0..\infty)$, such that d_i is the number of copies of item i "still under way". The channel then is assumed to deliver only copies of items i for which d_i is positive, and with each reception of a copy of item i array element d_i is decreased by 1.

0.3.3 How (not) to model loss of items

A channel is allowed to not deliver any particular item, but to be able to guarantee progress we must, of course, restrain the amount of items lost appropriately. This can be achieved by associating, with each item delivered to the receiving end of

the channel, an arbitrary natural number that represents the number of items lost before this item is received.

More precisely, we identify the arriving items by means of consecutive numbering, counting from 0 onwards; to distinguish these numbers from the numbering used in the previous subsection, we call the current numbers "arrival numbers". So, the arrival number of an item is the number of items that have been received before it. With the item with arrival number m we associate a natural number C_m: this is the number of items lost just before arrival of item m. So, when the item with arrival number m is received a total of $m+1$ items really have been received and a total of $(\Sigma i : 0 \leq i \leq m : C_i)$ items have been lost; thus, a total of $(\Sigma i : 0 \leq i \leq m : C_i + 1)$ items have been sent, resulting in the reception of the first $m+1$ items.

Notice that this bounds the number of items sent per item received: as soon as the Sender has sent at least $(\Sigma i : 0 \leq i \leq m : C_i + 1)$ items at least $m+1$ will actually arrive. In this study, therefore, we will not use this formalization directly. Instead, we use as a consequence what we call the

Bounded Loss Assumption .

This assumption states that if the Sender continues to send items through the channel the Receiver will continue to receive items from the channel as well. That is, if progress of the Sender is guaranteed progress of the Receiver is guaranteed as well. Note that this is a direct consequence of the above formal rendering of item loss.

Finally, we emphasize that the above principle gives no information on the identities of the items lost or received. As a very simple example, consider the special case where $C_i = 1$, for all natural i, and consider the case where the Sender sends "red" items and "white" items strictly alternatingly. If the channel now delivers white items only, so all red items are lost, then this channel still satisfies the Bounded Loss Assumption: after all, it delivers every other item sent.

Of course, this phenomenon has grave consequences for the design of a reliable transmission protocol: the only way to guarantee that at least one red item is received is to design the protocol in such a way that, after a while, only red items are sent! Fortunately, all Sliding Window Protocols have this property.

0.4 Implementation issues

We recall that a bounded buffer can only *temporarily* bridge speed differences between a producer of items and a consumer of items: in the long run a producer and a consumer, even when connected via a bounded buffer, can only proceed at *the same average speed*. This is achieved by synchronization: the consumer must be synchronized to prevent it from taking items from an empty buffer, and the producer

must be synchronized to keep the number of items in the buffer bounded, that is, to prevent *buffer overflow*. This kind of synchronization sometimes is also referred to by the phrase "flow control".

A First-In First-Out buffer can most conveniently be represented by an infinite array of items $b[0..\infty)$, say, together with two indices p and q, say, satisfying the invariant:

$$0 \le p \le q \quad ,$$

such that $b[p..q)$ represents the actual content of the buffer. Adding an item x to this buffer then amounts to:

$$b[q] := x \ ; \ q := q+1 \quad ,$$

whereas $b[p]$ is the buffer's element to be taken first and where $p := p+1$ amounts to removing this first element from the buffer. Of course, both inspection and removal of the buffer's first element are subject to the precondition $p<q$: if $p=q$ the buffer is *empty*.

If the buffer is bounded it satisfies an additional invariant:

$$q \le p+N \quad ,$$

where N is the (finite) *capacity* of the buffer. In this case the infinite array can be implemented as a finite array $fb[0..N)$, say. After all, the number of relevant elements of $b[p..q)$ now is at most N, and these can be conveniently represented by fb as follows:

$$(\forall i : p \le i < q : b[i] = fb[i \bmod N]) \quad .$$

Now, the buffer's first element is $fb[p \bmod N]$ and the buffer's first empty slot is $fb[q \bmod N]$.

Such an implementation is known as a "cyclic array", and it is nice, but we take it for what it really is: just an implementation. In this study we will happily use infinite arrays to represent buffers and we will pay little attention to their finite implementations, by means of cyclic arrays or otherwise.

* * *

In the same vein, suppose we have an infinite boolean array $b[0..\infty)$, together with two natural variables p and q. Suppose the following invariants are given:

Q0: $0 \le p \le q$,

Q1: $(\forall : 0 \le i < p : b[i])$,

Q2 : $(\forall : q \leq i : \neg b[i])$.

This may be used, for example, to represent a finite subset of the naturals by means of the relation that every natural n is an element of the set if and only if $b[n]$.

To implement this we do not really need an infinite array: because of Q1 the segment $b[0..p)$ is not particularly interesting, as it can be compactly represented by variable p alone. Similarly, the segment $b[q..\infty)$ is not very interesting either, as it can be compactly represented by q alone. What remains is the segment $b[p..q)$, which can be stored in a finite array $c[0..r)$, say, where $r = q - p$. From, p, q, and c the elements of b can be reconstructed, as follows, for any natural n:

$$\begin{array}{lll} b[n] = \mathsf{true} & , \text{ if } & n < p \\ b[n] = c[n-p] & , \text{ if } & p \leq n < q \\ b[n] = \mathsf{false} & , \text{ if } & q \leq n \end{array}$$

Again, we consider this as an implementation issue of minor importance, and we will happily use infinite arrays like b as we see fit.

0.5 Real-time issues

In our presentation of the design all components receiving items from communication channels will have the following general structure:

$*[$ receive x
; "process x"
$]$

Here the part "process x" will contain no blocking statements in general and, hence, no other receive statements in particular. As a result, such a component is only synchronized with its environment via the (one-and-only) "receive x" statement.

If the communication channel via which x is received delivers items faster than (the execution of) the component can consume them, some items may be lost and/or other errors may occur. Such errors can only be prevented, either by assuming the presence in the communication channel of sufficient buffering of items that cannot yet be received by the component, or by imposing the *real-time requirement* that the component is executed fast enough.

The latter means that, actually, the component is always ready to perform its (first or next) "receive x" statement *before* arrival of the (first or next) item from

the channel. In low-level implementations this real-time requirement can be met by implementing the component as a, so-called, *interrupt-service routine*, which is executed with very high priority: as soon as the communication channel presents an item for reception, an interrupt is triggered, and the corresponding interrupt-service routine takes care of actually receiving that item. This modus operandi is only feasible, of course, if the amount of work involved with consuming item x is small enough so as not to disturb the scheduling of regular activities inside the executing machine too much.

In our presentation of the design we will see to it that this is the case, and we will not address the issue of this real-time rqeuirement anymore: we just assume that the components receiving items from the communication channels are fast enough, and we do not assume the presence of buffers at the receiving ends of the channels. Any buffering required for the sake of the protocol will be programmed explicitly, including the flow control needed to keep the amount of items in these buffers bounded. In addition, in our presentation of the design the parts "process x" in the receiving components will essentially entail little more than recording the event that an item has been received (and which).

Finally, we remark that we can safely use a send-statement in the part "process x", because we have assumed send-statements to be non-blocking. Thus, the occurrence of an occasional send-statement in such a part will not gravely affect the real-time obligations associated with it.

Chapter 1

First Approximation

1.0 The Sending Client and the Send Buffer

At the sending side we model the interface with the protocol as a separate component, called "Sending Client", which has the following structure – explanation follows –:

$$
\begin{array}{l}
*\,[\ \text{“produce item } x\text{”} \\
\ \ ;\ X[\,sq\,] := x\ ;\ sq := sq{+}1 \\
\ \ ]
\end{array}
$$

Each time Sending Client has "produced" an item it offers this item for transmission by storing it in infinite array $X[\,0\,..\,\infty\,)$ at its first free position, as represented by index variable sq. (Notice that, in the above code, x just is a local variable of the repetition's body.) Thus, array segment $X[\,0\,..\,sq\,)$ represents all items that ever have been offered for transmission. Initially, no items have been offered, hence the initial value of sq is 0, and an obvious system invariant is $0 \leq sq$.

The protocol will implement reliable transmission of the elements of $X[\,0\,..\,sq\,)$ via the unreliable communication channel. As a result of this activity, at any moment in time *some* of X's elements will have been transmitted successfully whereas *others* have not been transmitted successfully (yet). The elements that have been transmitted successfully will, in general, not correspond to a consecutive segment of X, but we will see that it nevertheless turns out useful to introduce an additional index variable sp, with invariant:

Q0: $\quad 0 \leq sp \leq sq$.

The initial value of sp must be 0 too, and the intended interpretation now is that at least all elements of $X[\,0\,..\,sp\,)$ (and perhaps some more) have been transmitted

successfully via the channel. As a result, the items in this array segment are not relevant anymore for the protocol and, therefore, they need not be retained.

Therefore, we consider array segment $X[sp..sq)$ as the "Send Buffer". Unless Sending Client is appropriately synchronized, this buffer is in principle unbounded. If so desired the buffer can be bounded by strengthening invariant Q0, thus:

Q0a: $0 \leq sp \leq sq \leq sp+W$,

where constant W is the Send Buffer's capacity. To guarantee the invariance of Q1 Sending Client must be properly synchronized: the required precondition of "$X[sq] := x$; $sq := sq+1$" then is: $sq < sp+W$. We do not consider bounding the Send Buffer as specific to the Sliding Windows Protocol but, as we will see later, we will need the stronger invariant Q0a for the sake of the protocol anyway, and in what follows W will be called the "Window Size". Constant W is assumed to satisfy: $1 \leq W$. The protocol will only transmit items from array segment $X[sp..sp+W)$; as we will see, this restriction turns out necessary to guarantee *progress*. By incorporating this we obtain this program for

Sending Client:

$*[$ "produce item x"
; est $sq < sp+W$
; $\{ 0 \leq sp \leq sq < sp+W \}$
$X[sq] := x$; $sq := sq+1$
$]$

warning: As a matter of fact we are discussing two essentially *different* problems here, whose solutions not necessarily coincide. On the one hand there is no point in having a Window Size that exceeds the capacity of the Send Buffer, but all by itself there is no reason why the two should be equal. It is perfectly conceivable to have a Send Buffer whose capacity is larger than the Window Size. In such an arrangement, Sending Client is able to buffer more items for transmission than the protocol will actually transmit.

□

1.1 The Receiving Client and the Receive Buffer

At the receiving side we model the interface with the protocol as a separate component, called "Receiving Client", with the following structure – explanation follows –:

$*[$ est $rq < sq$
; $\{\ 0 \le rq < sq\ \}$
$Z[rq] := X[rq]\ ;\ rq := rq+1$
; "private consumer activity"
$]$

Array segment $Z[0..rq)$ represents all items delivered to Receiving Client, and the statement "$Z[rq] := X[rq]\ ;\ rq := rq+1$" represents delivery of the next item. Initially no items have been delivered at all, so initially $rq = 0$.

Array Z is only used to model delivery adequately; thus, $Z[0..rq)$ represents the complete delivery history of the protocol. In any concrete implementation, of course, array Z may not be required: it is up to the designer of Receiving Client to decide how delivered items will be consumed further.

Because item delivery cannot run ahead of sending the items by the Sending Client, an obvious invariant must be:

Q1: $\quad 0 \le rq \le sq$,

which gives rise to $rq < sq$ as a precondition, and as a guard, to the delivery operation. Hence, the synchronization by means of "est $rq < sq$" is unavoidable: this is part of the problem and not specific to the Sliding Windows Protocol.

By construction, Receiving Client also maintains this invariant:

Q2: $\quad (\forall i : 0 \le i < rq : Z[i] = X[i]\ )$,

which expresses that all delivered items have been sent (previously) by Sending Client. In addition, this guarantees First-In First-Out delivery as well: Receiving Client initializes the elements of array Z in precisely the same order as Sending Client stores them in array X.

1.2 A progress requirement

Together Sending Client and Receiving Client correctly implement FIFO-transmission of items from the former to the latter. In reality, however, the system is distributed and neither client can contain references to variables of the other; that is, the use of shared variables is prohibited and all communication will have to take place via the unreliable channels.

In particular, the guard $rq < sq$, in Receiving Client, cannot be implemented directly, as it contains variable sq, which is private to Sending Client. Therefore, this guard will have to be strengthened to a, yet to be determined, boolean expression rB, say, thus:

$*[$ { **??** $rB \Rightarrow rq < sq$ }
  est rB
; { $0 \leq rq < sq$ }
  $Z[rq] := X[rq]$; $rq := rq+1$
; "private consumer activity"
]

Because the new guard will be stronger than the old one, correctness of the annotation is preserved, but now we are faced with the following *progress requirement*:

R0 : $rq < sq \;\Rightarrow$ "after finitely many steps of the system, rB is true" .

Notice that states where $rq = sq$ constitute no problem: as we have seen already, Receiving Client becoming blocked in such states not only is harmless but even is *unavoidable*: $rq = sq$ means that all items offered by Sending Client have been delivered to Receiving Client, hence progress of Receiving Client *must* become impossible.

To guarantee true progress, that is, to guarantee absence of (the danger of) individual starvation, the guard rB must not only become true but it must become *stable* as well. We will see to it, however, that *all* guards in the protocol are stable: once they have made the transition from false to true they will remain true forever. Notice, by the way, that the original weaker guard, $rq < sq$, is stable indeed.

1.3 The Receive Buffer

Both index variable sq and array X are variables of the Sender, hence they must be eliminated from Receiving Client. For this purpose the expression $X[rq]$ must be replaced by a different but equivalent expression. This will require communication with the Sender, as only the Sender can provide items from array X.

In view of the unreliability of the communication channels and in view of the fact that the channels do not preserve the order of transmitted items, we must allow for the possibility that items arrive at the Receiver that are not $X[rq]$. Hence, some form of buffering at the Receiver is needed, for the purpose of storing items that arrive out of order.

We represent this buffer as an infinite array $Y[0..\infty)$ of items, together with a boolean array $rc[0..\infty)$, that satisfy the following invariant:

Q3 : $(\forall i :: rc[i] \Rightarrow Y[i] = X[i])$.

This invariant expresses that the elements of Y for which the corresponding elements of rc are true are equal to the corresponding elements of X. Boolean

array rc is necessary here because, due to the irregularities in the channel, we may not expect the content of the Receive Buffer to fill a consecutive segment in array Y. Thus, boolean array rc represents the set of all indices i for which $Y[i] = X[i]$ is guaranteed. Initially, invariant Q3 holds, of course, provided that initially: $(\forall i :: \neg rc[i])$.

In the above program for Receiving Client we are only interested in $X[rq]$. In view of the condition $rq < sq$ and invariant Q1, only the values of $rc[i]$ for $i < sq$ will be of interest. This is also in line with the operational interpretation of the game: boolean $rc[i]$ indicates whether a copy of array element $X[i]$ is available in the Receive Buffer, and for $sq \leq i$ this will never be the case, because for $sq \leq i$ array element $X[i]$ has not even been initialized!

Therefore, it is safe to introduce the following additional invariant:

Q4: $\quad (\forall i :: sq \leq i \Rightarrow \neg rc[i]) \quad ,$

which trivially holds initially, because of the initial value of rc. By the instantiation $i := sq$ we conclude that Q4 implies $\neg rc[sq]$ as a special case.

On account of Q3, array element $X[rq]$ now is equal to $Y[rq]$, provided $rc[rq]$ is true; in addition we derive:

$$
\begin{array}{ll}
 & rc[rq] \\
\Rightarrow & \quad \{\ \text{Q4, with contraposition}\ \} \\
 & rq < sq \quad ,
\end{array}
$$

so for the still to be chosen expression rB we can use $rc[rq]$, as this implies $rq < sq$, as required. Initially, the elements of rc are false and once they have become true they will remain true forever. Hence, the guard $rc[rq]$ is stable.

Thus, by substituting these results we obtain the following new version of the program for

Receiving Client:

$$
\begin{array}{l}
*[\ \mathsf{est}\ rc[rq] \\
\ ;\ \{\ rc[rq]\ ,\ \text{hence: } 0 \leq rq < sq\ \wedge\ Y[rq] = X[rq]\ \} \\
\quad Z[rq] := Y[rq] \\
\ ;\ rq := rq + 1 \\
\ ;\ \text{“private consumer activity”} \\
\ ]
\end{array}
$$

With this choice for the guard rB our original progress requirement R0 can now be reformulated:

R1: $rq < sq \;\Rightarrow$ "after finitely many steps of the system, $rc[rq]$ is true" .

We still have to decide how arrays rc and Y are to obtain their values. We have already decided that all elements of rc are false initially; in addition, we will see to it that every element of rc makes the transition to true exactly once, and once it is true it will remain true. Hence, the guard $rc[rq]$ is stable.

Because the assignment $rq := rq+1$ has $rc[rq]$ as its precondition, we obtain yet another invariant for free, namely:

Q5: $(\forall i : i<rq : rc[i])$.

Together, arrays rc and Y constitute the Receive Buffer. On account of invariant Q3, the buffer positions i with $sq \leq i$ are (still) empty. On account of Q4 and because of the order in which the elements of Y are copied into Z, the contents of the buffer positions i with $i < rq$ may be considered (no longer) relevant; hence, these positions can be considered empty (again) as well.

As a consequence, the contents of the Receive Buffer is confined to the array segments $rc[rq..sq)$ and $Y[rq..sq)$. So, the Receive Buffer will actually contain at most $sq-rq$ items.

Finally, in the previous section we have introduced (in the Sender) a variable sp, with the intended interpretation that at least all elements of $X[0..sp)$ have been "transmitted successfully". This can now be defined as "being present in the Receive Buffer", that is, element $X[i]$ has been "transmitted successfully" if and only if $rc[i]$ and, hence, $Y[i]=X[i]$. So, the intended meaning of variable sp is captured adequately by the following additional invariant:

Q6: $(\forall i : i<sp : rc[i])$.

This invariant also tells us when sp may be incremented, namely when $rc[sp]$ is true. As there is no reason why incrementing sp should be synchronous with the activity of Sending Client, we introduce a dedicated parallel component, in the Sender, for this purpose, called "Co-Sender" – notice that $sp < sq$ is the required additional precondition for the invariance of Q0a–.

Co-Sender:

$*[$ est $rc[sp]$
; { $rc[sp]$, hence by Q4: $sp < sq$ }
$sp := sp+1$
]

This is a component in the Sender, because sp is a Sender variable; as a result, variable rc, which is a Receiver variable, may not be used here. How this is to be implemented will be taken care of later, though.

1.4 The system thus far (i)

In its current state the design is still incomplete – it lacks progress –, but at least all its invariants and assertions are correct. Thus far the system consists of invariants Q0a up to Q5 and three components, namely the Sending and Receiving Clients and Co-Sender:

Q0a: $0 \le sp \le sq \le sp+W$

Q1: $0 \le rq \le sq$

Q2: $(\forall i : 0 \le i < rq : Z[i] = X[i])$

Q3: $(\forall i :: rc[i] \Rightarrow Y[i] = X[i])$

Q4: $(\forall i :: sq \le i \Rightarrow \neg rc[i])$

Q5: $(\forall i : i < rq : rc[i])$

Q6: $(\forall i : i < sp : rc[i])$

initially: $sp = 0 \ \wedge \ sq = 0 \ \wedge \ rq = 0 \ \wedge \ (\forall i :: \neg rc[i])$.

Sending Client:

$*[$ "produce item x"
; est $sq < sp+W$
; $\{ 0 \le sp \le sq < sp+W \}$
$X[sq] := x \ ; \ sq := sq+1$
$]$

Co-Sender:

$*[$ est $rc[sp]$
; $\{ rc[sp]$, hence by Q4: $sp < sq \}$
$sp := sp+1$
$]$

Receiving Client:

$*[$ est $rc[rq]$
; $\{ rc[rq]$, hence: $0 \le rq < sq \ \wedge \ Y[rq] = X[rq] \}$
$Z[rq] := Y[rq] \ ; \ rq := rq+1$
; "private consumer activity"
$]$

Progress requirement, yet to be satisfied:

R1: $rq < sq \ \Rightarrow$ "after finitely many steps of the system, $rc[rq]$ is true" .

Chapter 2

Forward Communication

2.0 The Receiver Proper

The current state of the design is such that we still have meet progress requirement R1: in states where $rq < sq$ the boolean $rc[rq]$ becomes true within finitely many steps of the system: as soon as $rc[rq]$ is true, Receiving Client can proceed and can consume its next item, by incrementing rq. Thus, completion of the design is mainly driven by this progress requirement.

For this purpose we introduce an additional parallel component in the Receiver, called "Receiver Proper", that will establish $rc[rq]$. Invariant Q3 now tells us that the assignment $rc[rq] := \mathsf{true}$ requires $Y[rq] = X[rq]$ as its precondition. The only way to establish this is to assign the value of $X[rq]$ to $Y[rq]$, and $X[rq]$ can only be obtained by receiving it from the Sender via the (forward) communication channel.

Because we have no control over the order in which sent items will be received via the communication channel we cannot[0] simply require that just item $X[rc]$ is received. So we have to allow for the possibility that *any* item $X[h]$, say, from the Send Buffer is received by Receiver Proper; if this happens this value can be assigned to $Y[h]$ and $rc[h]$ can be made true. Invariant Q4 requires that h satisfies $h < sq$, which, therefore, will be a postcondition of the receive statement.

In addition, we do not assume items to identify their own indices in array X – that is, their positions in the Send Buffer –; actually, different elements of X may even be equal: we do *not* assume items to be "self-identifying". Therefore, every item will be sent together with its *index* in X; so, the Sender will send *pairs* $\langle h, X[h] \rangle$, satisfying $0 \leq h < sq$. These considerations give rise to the following

[0] Well, we could, but this would lead to a straightforward hand-shake protocol, which would not be very efficient.

design for Receiver Proper:

$$
\begin{array}{l}
*[\ \ \mathsf{receive}\ \langle h, x\rangle \\
\ \ ;\ \{\ 0 \le h < sq\ \wedge\ x = X[h]\ \} \\
\qquad Y[h] := x\ ;\ rc[h] := \mathsf{true} \\
\ \ ]
\end{array}
$$

remark: This component is subject to the real-time requirement mentioned in Section 0.5; as its only job is to transfer received values to their proper places in the Receive Buffer – which, after all, is the net effect of the assignments "$Y[h] := x\ ;\ rc[h] := \mathsf{true}$" – this should pose no large difficulties.
□

Thus, Receiver Proper sets elements of boolean array rc to true, while respecting the system invariants. It has the *possibility* to set $rc[rq]$ to true as well, as required, but this is not (yet) guaranteed. To guarantee that $rc[rq]$ becomes true in finitely many steps, the Sender must be designed towards this goal: after all, to ensure progress in Receiver Proper we have to construct a matching component actually sending pairs received by Receiver Proper.

2.1 The Sender Proper

In the Sender we now introduce an additional parallel component, called "Sender Proper", the purpose of which is to send items from the Send Buffer, paired with their indices, via the communication channel. This must be organized in such a way that, eventually, item $X[rq]$ is guaranteed to be received by Receiver Proper.

By invariant Q3, there is no point in transmitting an item $X[k]$ for which $rc[k]$ already is true; therefore, we restrict transmission to indices satisfying $\neg rc[k]$. As a special case, by invariant Q6, there is no point in transmitting an item with an index less than sp; hence, we restrict the transmission of items to the ones with indices at least sp. Finally, the Send Buffer contains no items with indices at least sq[1]; so, we further restrict the items to those with indices less than sq.

Thus, we obtain the following design for Sender Proper – explanation follows –:

$$
\begin{array}{l}
*[\ \ k :\ sp \le k < sq\ \wedge\ \neg rc[k] \\
\ \ ;\ \{\ sp \le k < sq\ \} \\
\qquad \mathsf{send}\ \langle k, X[k]\rangle \\
\ \ ]
\end{array}
$$

[1] This is partially reflected by invariant Q1; a more complete formalization would involve an additional boolean array, to represent the set of items in the Send Buffer.

The statement $k : sp \le k < sq \wedge \neg rc[k]$ assigns a value to variable k such that $sp \le k < sq \wedge \neg rc[k]$; if $sp = sq$ this range is empty and no such value exists, so the statement is blocking the component: if $sp = sq$ the Send Buffer is empty and Sender Proper will become blocked until Sending Client offers more items for transmission: Sender Proper is synchronized with the Sending Client, as it should be.

If, however, $sp < sq$ and if, in addition, $(\forall i : sp \le i < sq : rc[i])$ then Sender Proper will become blocked too, but in this case all items in the Send Buffer have been successfully delivered to the Receive Buffer. Now component Co-Sender will proceed and will increment sp to (eventually) the current value of sq: a state satisfying $sp < sq \wedge (\forall i : sp \le i < sq : rc[i])$ is an (unstable) *transient state* that will not last forever.

2.2 Progress

Component Sender Proper has been constructed in such a way that no items will be selected for which already has been established that they have been transmitted successfully, as represented by boolean array rc. This turns out to be sufficient to guarantee progress: we can now prove that progress requirement R1 is satisfied.

To be able to formulate this proof we must represent the amount of duplication of items in the communication channel. For this purpose we introduce an auxiliary variable sn, to distinguish all individual send events, and a constant natural function $D[0..\infty)$, with the interpretation that D_i is the maximal number of copies, including the original, of the item sent in transmission i, for all $i : 0 \le i < sn$.

In addition we introduce an auxiliary array variable $sd[0..\infty)$, to represent the number of copies of a particular item "still under way" in the channel. Because pairs $\langle k, X[k] \rangle$ with the same index k cannot be distinguished, we do not, however, let sd_i represent the number of copies "still under way" of the pair sent in transmission i; instead, sd_k will be the total number of copies "still under way" of any pair $\langle k, X[k] \rangle$ with index k. Hence, instead of augmenting the statement $\mathsf{send}\ \langle k, X[k] \rangle$ with the obvious $sd_k := D_{sn}$ we decorate it with $sd_k := sd_k + D_{sn}$; in addition, we decorate the statement $\mathsf{receive}\ \langle h, x \rangle$, in Receiver Proper, with $sd_h := sd_h - 1$. These additions yield the following code for Sender Proper and Receiver Proper:

Sender Proper:

$$
\begin{array}{l}
*[\ \ k : sp \le k < sq \wedge \neg rc[k]\ \ ;\ \ sd_k := sd_k + D_{sn}\ \ ;\ \ sn := sn + 1 \\
\ \ ;\ \{\ sp \le k < sq\ \} \\
\qquad \mathsf{send}\ \langle k, X[k] \rangle \\
\ \ ]
\end{array}
$$

Receiver Proper:

$$
\begin{array}{l}
*\,[\ \ \mathsf{receive}\ \langle h, x\rangle\ ;\ \{\ 1 \le sd_h\ \}\ sd_h := sd_h - 1 \\
\quad ;\ \{\ 0 \le h < sq\ \wedge\ x = X[h]\ \} \\
\qquad Y[h] := x\ ;\ rc[h] := \mathsf{true} \\
\quad]
\end{array}
$$

Notice that, in Sender Proper, we have added the extra statements to the selection of k, instead of to the actual send-statement. This corresponds to a (harmless) act of clairvoyance: the premature increase of sd_k (with D_{sn}) can be viewed as a prediction of the amount of duplication the item will incur during its actual transmission. For the sake of the proof of progress, we consider the assignments to sd_k and sd_h as being indivisible parts of the immediately preceding statements.

Initially, all elements of sd are assumed to be 0, of course, and an obvious invariant is $(\forall i :: 0 \le sd_i)$.

* * *

Now we are ready for the proof that progress requirement R1 is satisfied. We consider states satisfying $rq < sq \wedge \neg rc[rq]$. It now suffices to show that as long as this condition holds the system's activity will eventually terminate, and that it will do so in a state where $rc[rq]$.

Because $\neg rc[rq]$ we conclude, by invariant Q6, that $sp \le rq$; hence, rq is in the range of the selection $k : sp \le k < sq \wedge \neg rc[k]$ in Sender Proper. So, this selection is non-blocking and Sender Proper will proceed indefinitely – that is, as long as $rq < sq \wedge \neg rc[rq]$ remains true –. By the Bounded Loss Assumption this means that Receiver Proper will proceed indefinitely too.

Now, the pair – with lexicographic ordering –:

$$\langle\ (\#\, i : i < rq + W : \neg rc[i])\ ,\ (\Sigma i : rc[i] : sd_i)\ \rangle$$

is a useful variant function for the communication channel. Receiver Proper repeatedly receives pairs $\langle h, x\rangle$, and here two cases can be distinguished:

- If $\neg rc[h]$ the subsequent assignment $rc[h] := \mathsf{true}$ decreases the first element of the variant function by 1; this assignment also increases the pair's second element by sd_h but this is harmless in view of the lexicographic ordering.
- If $rc[h]$ then the first element of the variant function remains the same, and its second element decreases by 1, because the reception of the pair $\langle h, x\rangle$ always is accompanied by a decrease of sd_h by 1: hence, because $rc[h]$ the sum $(\Sigma i : rc[i] : sd_i)$ decreases by 1 too.

Moreover, no other statement in the above program increases the variant function. In particular, the only potentially conflicting statement is $sd_k := sd_k + D_{sn}$ in Sender Proper, but this statement has $\neg rc[k]$ as its precondition, hence this assignment to sd_k does not affect the second element of the variant function.

* * *

Finally, we observe that, without decreases of the first element of the variant function, its second element, $(\Sigma i : rc[i] : sd_i)$, can only be decreased a finite amount of times. Hence, the variant function's first element, $(\#\, i : i < rq+W : \neg rc[i])$, must be decreased eventually as well. Hence, as long as the required condition, $rq = sq \vee rc[rq]$, has not been achieved, the number of indices for which $\neg rc[i]$ will continue to be decreased, thus eventually leading to a state satisfying $rc[rq]$, because we have:

$$(\#\, i : i < rq+W : \neg rc[i]) = 0 \quad \Rightarrow \quad rc[rq] \quad .$$

2.3 The system thus far (ii)

Q0a : $0 \leq sp \leq sq \leq sp+W$

Q1 : $0 \leq rq \leq sq$

Q2 : $(\forall i : 0 \leq i < rq : Z[i] = X[i])$

Q3 : $(\forall i :: rc[i] \Rightarrow Y[i] = X[i])$

Q4 : $(\forall i :: sq \leq i \Rightarrow \neg rc[i])$

Q5 : $(\forall i : i < rq : rc[i])$

Q6 : $(\forall i : i < sp : rc[i])$

initially: $sp = 0 \ \wedge \ sq = 0 \ \wedge \ rq = 0 \ \wedge \ (\forall i :: \neg rc[i]) \ \wedge$
$sn = 0 \ \wedge \ (\forall i :: sd_i = 0) \quad .$

Sending Client:

```
* [  "produce item x"
  ;  est  sq < sp+W
  ;  { 0 ≤ sp ≤ sq < sp+W }
     X[sq] := x ; sq := sq+1
  ]
```

Co-Sender:

$*[$ est $rc[sp]$
; $\{\ rc[sp]$, hence by Q4: $sp < sq\ \}$
$sp := sp+1$
$]$

Sender Proper:

$*[\ k : sp \le k < sq\ \wedge\ \neg rc[k]\ ;\ sd_k := sd_k + D_{sn}\ ;\ sn := sn+1$
; $\{\ sp \le k < sq\ \}$
send $\langle\, k, X[k]\,\rangle$
$]$

Receiving Client:

$*[$ est $rc[rq]$
; $\{\ rc[rq]$, hence: $0 \le rq < sq\ \wedge\ Y[rq] = X[rq]\ \}$
$Z[rq] := Y[rq]$; $rq := rq+1$
; "private consumer activity"
$]$

Receiver Proper:

$*[$ receive $\langle h, x\rangle$; $\{\ 1 \le sd_h\ \}\ sd_h := sd_h - 1$
; $\{\ 0 \le h < sq\ \wedge\ x = X[h]\ \}$
$Y[h] := x$; $rc[h] :=$ true
$]$

Chapter 3

Backward Communication

3.0 Acknowledgements

In the current state, the design is correct and progress is guaranteed, but in two of the Sender's components, namely Sender Proper and Co-Sender, the Receiver's variable rc still occurs. To obtain a truly distributed implementation this variable must be eliminated from these components.

To do so we introduce a new Sender variable, sc, also an infinite boolean array, that will take over the role of rc in the Sender. The relation between sc and rc is given by the following new invariant:

Q7: $\quad (\forall i :: sc[i] \Rightarrow rc[i]) \quad .$

Invariant Q7 is satisfied initially if initially $(\forall i :: \neg sc[i])$.

Now we strengthen the guard $rc[sp]$ in Co-Sender to $sc[sp]$, thus retaining its correctness and eliminating variable rc. Also, we replace $\neg rc[k]$ in Sender-Proper by $\neg sc[k]$. This effectively *weakens* the guard it is part of, so, we will have to reconsider the progress argument.

Also for the sake of progress we must provide for a way to set the elements of sc to true. Because of invariant Q7 every assignment $sc[l] := \mathsf{true}$ has $rc[l]$ as its precondition. To establish this we use dedicated items, via the backward channel from the Receiver to the Sender, to communicate exactly this fact. These items are called "acknowledgements"; every such acknowledgement contains an index identifying an element of array rc.

Because arrivals (at the Sender) of acknowledgements are asynchronous events we introduce yet another dedicated component, "Co-Sender 1", to process them. Arrival of an acknowledgement containing index l signals that $rc[l]$ is true; hence, after this arrival, array element $sc[l]$ may be set to true as well.

An acknowledgement with index h may only be sent whenever $rc[h]$ is true. Because the only assignment to rc occurs in Receiver-Proper, we add a statement "$\mathsf{send\text{-}ack}\ h$" right after the assignment $rc[h] := \mathsf{true}$.

remark: This is a true design decision: once $rc[h]$ has become true as many acknowledgements with index h may be sent as we like. Notice, however, that we are dealing with conflicting requirements here. On the one hand, sending acknowledgements with index h with as high a frequency as possible guarantees the swift arrival of an ackowledgement with index h at Co-Sender 1; on the other hand, this may unnecessarily consume capacity of the backward communication channel needed for acknowledgements with *other* indices (or needed for any other purposes, for that matter).

Moreover, for any particular index h we wish the stream of acknowledgements with index h to terminate. In the current design, one acknowledgement is sent per pair received via the forward channel: thus the number of acknowledgements with index h is *at most* the number of pairs $\langle h, x\rangle$ received. If the latter is finite so will be the former.

□

These additions lead to the following new components Sender-Proper, Co-Sender, Co-Sender 1, and Receiver-Proper:

Sender Proper:

$$
\begin{array}{l}
*[\ \ k : sp \le k < sq \ \wedge\ \neg sc[k] \\
\ \ ;\ \{\ sp \le k < sq\ \} \\
\ \ \ \ \mathsf{send}\ \langle k, X[k]\rangle \\
\ \ ]
\end{array}
$$

Co-Sender:

$$
\begin{array}{l}
*[\ \ \mathsf{est}\ \ sc[sp] \\
\ \ ;\ \{\ sc[sp]\ ,\ \text{hence, by Q7 and Q4: } rc[sp] \wedge sp < sq\ \} \\
\ \ \ \ sp := sp+1 \\
\ \ ]
\end{array}
$$

Co-Sender 1:

$$
\begin{array}{l}
*[\ \ \mathsf{receive\text{-}ack}\ l \\
\ \ ;\ \{\ 0 \le h < sq \ \wedge\ rc[l]\ \} \\
\ \ \ \ sc[l] := \mathsf{true} \\
\ \ ]
\end{array}
$$

Receiver Proper:

$$
\begin{array}{l}
*[\ \ \mathsf{receive}\ \langle h, x\rangle \\
\ \ ;\ \{\ 0 \le h < sq\ \wedge\ x = X[h]\ \} \\
\qquad Y[h] := x\ ;\ rc[h] := \mathsf{true} \\
\ \ ;\ \{\ 0 \le h < sq\ \wedge\ rc[h]\ \} \\
\qquad \mathsf{send\text{-}ack}\ h \\
\ \ ]
\end{array}
$$

3.1 A minor improvement?

In view of the real-time requirements associated with processing arriving acknowledgements component Co-Sender 1 is attractive: the required "processing" per received acknowledgement consists of the single assignment $sc[l] := \mathsf{true}$ only.

As this is the only assignment setting elements of sc to true, and as it is component Co-Sender's purpose to perform assignments $sp := sp+1$ with precondition $sc[sp]$, components Co-Sender and Co-Sender 1 can be combined into a single component, called "Combined Co-Sender", – provided this does not unacceptably aggravate the real-time obligations just mentioned –, as follows:

$$
\begin{array}{l}
*[\ \ \mathsf{receive\text{-}ack}\ l \\
\ \ ;\ \{\ 0 \le l < sq\ \wedge\ rc[l]\ \} \\
\qquad sc[l] := \mathsf{true} \\
\ \ ;\ \mathsf{do}\ sc[sp]\ \rightarrow\ \{\ sc[sp]\ ,\ \text{hence, by Q7 and Q4:}\ rc[sp] \wedge sp < sq\ \} \\
\qquad\qquad\qquad\qquad sp := sp+1 \\
\qquad \mathsf{do} \\
\ \ ]
\end{array}
$$

Notice that the use of a repetition is necessary here, because of the possibly irregular order of arrival of items: arrival of a single acknowledgement may give rise to a large increase of variable sp; this repetition takes over the role of the original repetition in component Co-Sender.

On account of invariant Q0a execution of the above repetition terminates after at most $sq - sp$ steps; by virtue of the same invariant this is at most W. Thus, the (worst-case) amount of work needed to process any received acknowledgement is at most W, which is more than the single assignment in the previous version but which still is bounded. (And, of course, on the average sp is increased by 1 per newly received acknowledgement.)

3.2 Progress

As in the previous chapter, we consider system states satisfying $rq < sq \wedge \neg rc[rq]$. As before, this and the invariants then imply $sp \leq rq < sq \wedge \neg sc[rq]$, so rq is in the range of the selection $k : sp \leq k < sq \wedge \neg sc[k]$ in Sender Proper. Hence, this selection is non-blocking and Sender Proper will proceed indefinitely. By the Bounded Loss Assumption Receiver Proper will proceed indefinitely as well, and for every pair thus received Receiver Proper sends an acknowledgement back to the Sender; once more by the Bounded Loss Assumption but now for the backward communication channel, component Combined Co-Sender will proceed indefinitely too.

remark: Notice that here we use that a cascade of two unreliable communication channels is equivalent to a single unreliable communication channel.
□

Therefore, it suffices to construct a variant function and to show that the actions of Combined Co-Sender effectively decrease the varient function, and that no other action increases it.

As before, for his purpose we introduce auxiliary variables to represent, for every index, the number of pairs with that index "still under way" in the forward communication channel, and also the number of acknowledgements with that index "still under way" in the backward communication channel. Constant natural functions D and C are used to model the amounts of duplication in the forward and backward communication channels respectively. This yields the following decorated components.

Q0a : $0 \leq sp \leq sq \leq sp+W$

Q1 : $0 \leq rq \leq sq$

Q2 : $(\forall i : 0 \leq i < rq : Z[i] = X[i])$

Q3 : $(\forall i :: rc[i] \Rightarrow Y[i] = X[i])$

Q4 : $(\forall i :: sq \leq i \Rightarrow \neg rc[i])$

Q5 : $(\forall i : i < rq : rc[i])$

Q6 : $(\forall i : i < sp : rc[i])$

Q7 : $(\forall i :: sc[i] \Rightarrow rc[i])$

initially: $sp = 0 \;\wedge\; sq = 0 \;\wedge\; rq = 0 \;\wedge\; (\forall i :: \neg rc[i]) \;\wedge\; (\forall i :: \neg sc[i]) \;\wedge\; sn = 0 \;\wedge\; (\forall i :: sd_i = 0) \;\wedge\; rn = 0 \;\wedge\; (\forall i :: rd_i = 0)$.

Sending Client:

$$
\begin{array}{l}
*[\ \text{“produce item } x\text{”} \\
\ ;\ \mathsf{est}\ \ sq < sp+W \\
\ ;\ \{\ 0 \le sp \le sq < sp+W\ \} \\
\quad X[\,sq\,] := x\ ;\ sq := sq+1 \\
\ ]
\end{array}
$$

Sender Proper:

$$
\begin{array}{l}
*[\ \ k :\ sp \le k < sq\ \wedge\ \neg sc[\,k\,]\ \ ;\ \ sd_k := sd_k + D_{sn}\ \ ;\ \ sn := sn+1 \\
\ ;\ \{\ sp \le k < sq\ \} \\
\quad \mathsf{send}\ \langle\, k\,, X[\,k\,]\,\rangle \\
\ ]
\end{array}
$$

Combined Co-Sender:

$$
\begin{array}{l}
*[\ \mathsf{receive\text{-}ack}\ l\ ;\ \{\ 1 \le rd_l\ \}\ rd_l := rd_l - 1 \\
\ ;\ \{\ 0 \le l < sq\ \wedge\ rc[\,l\,]\ \} \\
\quad sc[\,l\,] := \mathsf{true} \\
\ ;\ \mathsf{do}\ sc[\,sp\,] \rightarrow \{\ sc[\,sp\,]\ ,\ \text{hence, by Q7 and Q4: } rc[\,sp\,] \wedge sp < sq\ \} \\
\qquad\qquad\qquad\quad sp := sp+1 \\
\quad \mathsf{do} \\
\ ]
\end{array}
$$

Receiving Client:

$$
\begin{array}{l}
*[\ \mathsf{est}\ \ rc[\,rq\,] \\
\ ;\ \{\ rc[\,rq\,]\ ,\ \text{hence: } 0 \le rq < sq\ \wedge\ Y[\,rq\,] = X[\,rq\,]\ \} \\
\quad Z[\,rq\,] := Y[\,rq\,]\ ;\ rq := rq+1 \\
\ ;\ \text{“private consumer activity”} \\
\ ]
\end{array}
$$

Receiver Proper:

$$
\begin{array}{l}
*[\ \mathsf{receive}\ \langle h, x\rangle\ ;\ \{\ 1 \le sd_h\ \}\ sd_h := sd_h - 1\ \ ;\ \ rd_h := rd_h + C_{rn}\ \ ;\ \ rn := rn+1 \\
\ ;\ \{\ 0 \le h < sq\ \wedge\ x = X[\,h\,]\ \} \\
\quad Y[\,h\,] := x\ ;\ rc[\,h\,] := \mathsf{true} \\
\ ;\ \{\ 0 \le h < sq\ \wedge\ rc[\,h\,]\ \} \\
\quad \mathsf{send\text{-}ack}\ h \\
\ ]
\end{array}
$$

In terms of these auxiliary variables, we now use the following triple – with lexicographical ordering, as before – as variant function; the third element in this triple now accounts for (copies of) acknowledgements "still under way" in the backward channel:

$$\langle\ (\#\, i : i{<}rq{+}W : \neg\, sc[\,i\,]\,)\ ,\ (\Sigma i : sc[\,i\,] : sd_i\,)\ ,\ (\Sigma i : sc[\,i\,] : rd_i\,)\ \rangle\quad .$$

We now consider the following statements and their influence on the value of this variant function; the first two cases pertain to Combined Co-Sender:

- receive-ack l , while $\neg\, sc[\,l\,]$: here the subsequent assignment $sc[\,l\,] := \mathsf{true}$ decreases the first element of the variant function, thus decreasing the value of the variant function, as required.

- receive-ack l , while $sc[\,l\,]$: here the assignment $rd_l := rd_l - 1$ decreases the third element of the variant function, while leaving the first two elements unchanged, thus decreasing the value of the variant function, as required.

- receive $\langle h, x\rangle$, while $\neg\, sc[\,h\,]$: the assignments to neither $sd[\,h\,]$ nor $rd[\,h\,]$ change the value of the variant function.

- receive $\langle h, x\rangle$, while $sc[\,h\,]$ (and, hence, also $rc[\,h\,]$) : the second element of the variant function decreases due to the assignment $sd_h := sd_h - 1$, whereas its first element remains unchanged.

- $sd_k := sd_k + D_{sn}$ (in Sender Proper) : because this assignment has precondition $\neg\, sc[\,k\,]$ it does not influence the value of the variant function.

As in the previous Chapter, the value of the variant function cannot be decreased indefinitely: eventually, a state will emerge satisfying $\neg\, sc[\,rq\,]$, because:

$$(\#\, i : i{<}rq{+}W : \neg\, sc[\,i\,]\,) = 0 \quad \Rightarrow \quad sc[\,rq\,]\quad .$$

3.3 The final solution

Omitting the auxiliary variables introduced for the sake of the correctness proof, we obtain the following set of components implementing a Sliding Window Protocol.

Q0a : $\quad 0 \le sp \le sq \le sp{+}W$

Q1 : $\quad 0 \le rq \le sq$

Q2 : $\quad (\forall i : 0{\le}i{<}rq : Z[\,i\,] = X[\,i\,]\,)$

Q3 : $(\forall i :: rc[i] \Rightarrow Y[i] = X[i])$

Q4 : $(\forall i :: sq \leq i \Rightarrow \neg rc[i])$

Q5 : $(\forall i : i < rq : rc[i])$

Q6 : $(\forall i : i < sp : rc[i])$

Q7 : $(\forall i :: sc[i] \Rightarrow rc[i])$

initially: $sp = 0 \ \wedge \ sq = 0 \ \wedge \ rq = 0 \ \wedge \ (\forall i :: \neg rc[i]) \ \wedge \ (\forall i :: \neg sc[i])$.

Sending Client:

$*[$ "produce item x"
; est $sq < sp + W$
; $\{ \ 0 \leq sp \leq sq < sp + W \ \}$
$X[sq] := x \ ; \ sq := sq + 1$
$]$

Sender Proper:

$*[$ $k : sp \leq k < sq \ \wedge \ \neg sc[k]$
; $\{ \ sp \leq k < sq \ \}$
send $\langle k, X[k] \rangle$
$]$

Combined Co-Sender:

$*[$ receive-ack l
; $\{ \ 0 \leq l < sq \ \wedge \ rc[l] \ \}$
$sc[l] := \mathsf{true}$
; do $sc[sp] \ \rightarrow \ \{ \ sc[sp]$, hence, by Q7 and Q4 : $rc[sp] \wedge sp < sq \ \}$
$sp := sp + 1$
do
$]$

Receiving Client:

$*[$ est $rc[rq]$
; $\{ \ rc[rq]$, hence: $0 \leq rq < sq \ \wedge \ Y[rq] = X[rq] \ \}$
$Z[rq] := Y[rq] \ ; \ rq := rq + 1$
; "private consumer activity"
$]$

Receiver Proper:

$$
\begin{array}{l}
*[\ \ \mathsf{receive}\ \langle h, x\rangle \\
\ \ ;\ \{\ 0 \le h < sq\ \wedge\ x = X[h]\ \} \\
\qquad Y[h] := x\ ;\ rc[h] := \mathsf{true} \\
\ \ ;\ \{\ 0 \le h < sq\ \wedge\ rc[h]\ \} \\
\qquad \mathsf{send\text{-}ack}\ h \\
\ \ ]
\end{array}
$$

Chapter 4

Epilogue

4.0 What we have learned

Firstly, it proves definitely faesible to design protocols like the Sliding Window Protocol in a systematic way, along the lines of an Owicki-Gries style of reasoning, and as developed into a discipline in [2]. The use of *variant functions* in progress discussions turns out to be effective. As a matter of fact, the progress arguments happened to be smoother and simpler than I expected at the outset. As such, this was an encouraging experience.

The use of asynchronous communication poses no particular problems; in this respect this study distinguishes itself from [7], where synchronous communication is assumed.

An important aspect in presentations of more complicated designs is the use of nomenclature: what to name and, more importantly, what *not* to name. It is possible, for instance, to model the operation of the communication channels by assigning ordinal numbers to messages sent, thus modelling the complete history of the channels. It remains to be seen, however, to what extent this additional information contributes to the clarity of exposition. Also, for the sake of simplicty and clarity, we have deliberately chosen to use the Bounded Loss Assumption – see Section 0.3.3 – rather informally: complete formalization is tedious and adds little.

4.1 Still to be investigated

The indices used to identify both the items transmitted and the acknowledgements can be reduced modulo a constant that depends on the window size. This requires, however, additional assumptions on the behaviour of the communication channels, to the extent that messages travelling along the channels cannot be overtaken by

later messages indefinitely. First-In First-Out channels are a (very) special case of this. This has already been investigated in [8] and in [1], but a smooth development of this requires some more work. In [7] the channels are assumed to be FIFO, which makes the situation simpler.

In [7] the acknowledgements do not carry a single index number but a whole set of indices. This provides more information back to the Receiver, thus increasing the performance of the protocol but at the expense of larger messages used for acknowledgements. It remains to be investigated what the relative (dis)advantages of this variant are.

The version of the protocol presented here provides for unbounded buffering of items in Receiving Client, by means of array Z. Bounding the amount of items buffered here requires a normal form of flow control that involves extra acknowledgements back to the Sender. As this additional flow control is independent of the Sliding Window Protocol per se, so are the additional acknowledgements. Hence, the resulting system uses *two*, in principle unrelated, types of acknowledgements. Because, however, the protocol is asynchronous, its correct operation is insensitive to additional delays in the communication. Therefore, the two kinds of acknowledgements can be combined: one type of acknowledgement can serve both purposes. It remains to be investigated, though, what the consequences of such a design decision are. Can progress still be guaranteed? What is the, potentially negative, effect on the performance – read: throughput – of the system?

Bibliography

[1] R.E.J. de Backer, *A sliding-window protocol*.
master's thesis, Eindhoven University of Technology, 2001.

[2] W.H.J. Feijen, A.J.M. van Gasteren, *On a Method of Multiprogramming*.
Springer-Verlag, New York, 1999.

[3] R.R. Hoogerwoord, *A Formal Development of Distributed Summation*.
CS-Report 00-09, Eindhoven University of Technology, 2000.

[4] R.R. Hoogerwoord, *Leslie Lamport's Logical Clocks: a tutorial*.
CS-Report 02-01, Eindhoven University of Technology, 2002.

[5] A. van Leeuwen, *The sliding window protocol*.
master's thesis, Eindhoven University of Technology, 2006.

[6] S. Owicki, D. Gries, *An axiomatic proof technique for parallel programs I*. Acta Informatica **6**, pp. 319-340, 1976.

[7] J.L.A. van de Snepscheut, *The Sliding-Window Protocol Revisited*.
Formal Aspects of Computing **7**, pp. 3-17, 1995.

[8] N.V. Stenning, *A data transfer protocol*.
Computer Networks **1**, pp. 99-110, 1976.