Construction of Network Protocols by Stepwise Refinement*

A. Udaya Shankar
Department of Computer Science and
Institute for Advanced Computer Studies
University of Maryland
College Park, Maryland 20742

Simon S. Lam
Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712

Abstract. We present a heuristic to derive specifications of distributed systems by stepwise refinement. The heuristic is based upon a *conditional refinement* relation between specifications. It is applied to construct four sliding window protocols that provide reliable data transfer over unreliable communication channels. The protocols use modulo-N sequence numbers. They are less restrictive and easier to implement than sliding window protocols previously studied in the protocol verification literature.

Key words: Specification, refinement, sliding window protocols, transport protocols, distributed systems.

CONTENTS

1. Introduction
   1.1. Construction examples
   1.2. Organization of this report
2. Stepwise Refinement Heuristic
3. Sliding Window Protocol Construction: Initial Phase
   3.1. Initial system and requirements
   3.2. The sliding window mechanism
   3.3. Correct interpretation of data messages
   3.4. Correct interpretation of acknowledgement messages
   3.5. Progress requirement marking
4. Completing the Construction for Loss-only Channels
5. Completing the Construction for Loss, Reordering, and Duplication Channels
   5.1. Real-time system model
   5.2. A time constraint that enforces A_7
   5.3. A time constraint that enforces A_{10}
   5.4. Protocol I: implementation with 2N timers
   5.5. Protocol II: implementation with N timers
   5.6. Protocol III: implementation with one timer
6. Discussions
Tables 1-5
References

*The work of A. Udaya Shankar was supported by National Science Foundation under grant no. ECS-8502113 and grant no. NCR-8904590. The work of Simon S. Lam was supported by National Science Foundation under grant no. NCR-8613338 and by a grant from the Texas Advanced Research Program. This paper is an abbreviated version of [18].

1. Introduction

There are many ways to specify a distributed system. We advocate the following approach. Initially, a system is specified by a set of requirements, namely, desirable safety and progress properties that are expressed in some language. Subsequently, a specification of an implementation of the system is obtained in the form of a state transition system together with a set of fairness assumptions.[1] In general, it is quite difficult to derive the implementation specification from the requirements specification in one step. It is preferable to go through a succession of intermediate specifications, $\alpha_1, \alpha_2, \cdots, \alpha_n$, where each intermediate specification consists of a state transition system, a set of requirements and some fairness assumptions. In this paper, we present a stepwise refinement heuristic for constructing these specifications. The heuristic is based upon a weaker form of the refinement relation in [14], called *conditional refinement*, with the following property: α_{i+1} is a refinement of α_i if the heuristic terminates successfully.

At any point during a construction, we have a state transition system, a set of requirements, and a Marking. There are three types of requirements: *invariant requirements*, *event requirements* and *progress requirements*. The invariant and event requirements represent the safety properties desired of the system, and are specified by state formulas. Each event requirement is associated with a particular system event. The progress requirements are specified using *leads–to* assertions and fairness assumptions [14]. The Marking indicates the extent to which we have established that the requirements are satisfied by the specification.

We begin a construction with a set of state variables that provide just enough resolution in the system state space to *specify* the desired safety and progress properties of the distributed system. The desired safety properties are specified by invariant and event requirements. The desired progress properties are specified by progress requirements. None of the requirements are marked initially.

A succession of state transition systems is derived by applications of some *system refinement steps*. These steps increase the resolution of the system state space by adding new state variables, adding new messages, and refining a message into a set of messages. They change the set of state transitions by refining existing events and adding new events. We also apply some *requirement refinement steps* which can be used to strengthen the requirements. The objective of each refinement step is to increase the set of requirements that are marked. (Some of these refinement steps are illustrated in our construction of the sliding window protocols in Sections 3-5. A presentation of specific refinement steps is given in [18].)

The construction terminates successfully when all requirements are marked, and the nonauxiliary state variables and events satisfy the topology of the distributed system. The construction terminates unsuccessfully when a requirement is generated that is inconsistent with other requirements or with the initial condition of the system.

---

[1]If the state transition system is given in the relational notation, we refer to this as a *relational specification* [14].

1.1. Construction examples

Our heuristic is illustrated by a rigorous exercise in constructing four sliding window protocols that provide reliable data transfer between a producer and a consumer connected by unreliable channels. All protocols use modulo-N sequence numbers.[2] The desired property that sequence numbers in data messages and acknowledgement messages are interpreted correctly is stated as invariant requirements. We first construct a basic protocol that satisfies these *correct interpretation requirements* for channels that can only lose messages in transit. This basic protocol is then refined to be used for channels that can lose, duplicate and reorder messages arbitrarily. To satisfy the correct interpretation requirements for such channels, it is necessary that message lifetimes are bounded so that certain time constraints can be enforced in producing data blocks. We present three different ways of enforcing these time constraints, resulting in three protocols. The first and second of these protocols use $2N$ and N timers respectively. The third protocol uses a single timer to enforce a minimum time interval between producing successive data blocks. The minimum time interval is a function of N, the receive window size, and the maximum message lifetimes. To construct these three protocols, we use the system model developed in [16,17] in which real-time constraints can be specified and verified as safety properties.

To our knowledge, this is the first verified construction of sliding window protocols that use modulo-N sequence numbers where N is arbitrary. Our first and second protocols for loss, duplication and reordering channels appear to be novel. Our third protocol is best compared with the original Stenning's protocol [20]. Stenning verified certain safety properties assuming unbounded sequence numbers. He then informally argued that modulo-N sequence numbers can be used provided N satisfies a bound. His bound is similar to ours but not as tight as ours. Also, his protocol has several unnecessary requirements. (A detailed comparison is in Section 5.6.)

Knuth [11] has analyzed a sliding window protocol that uses modulo-N sequence numbers. He gives the minimum value of N that ensures correct data transfer for a special kind of channels, i.e., channels that can lose messages and allow messages to overtake a limited number of previously sent messages. Because of this restriction on the reordering of messages, his protocol does not require timers and the assumption of bounded message lifetimes.

In [19], we have extended the protocol for loss-only channels and the third protocol for loss, duplication and reordering channels to include the use of selective acknowledgement messages as well as variable windows for flow control.

1.2. Organization of this report

In Section 2, we give a brief description of our construction heuristic, including the conditional refinement relation between specifications. In Section 3, we derive the basic protocol and show that for channels that can lose, duplicate and reorder messages arbitrarily, its requirements are almost completely marked; only two invariant requirements concerning sequence numbers in channels

---

[2] In a real protocol, sequence numbers in data messages and acknowledgement messages are encoded by a small number of bits.

remain unmarked. In Section 4, we show that, for channels that can only lose messages, the basic protocol in fact satisfies all the requirements. In Section 5, we refine the basic protocol to obtain three different protocols that satisfy all the requirements for channels that can lose, duplicate and reorder messages arbitrarily. In Section 6, we discuss related work.

2. Stepwise Refinement Heuristic

The reader is assumed to be familiar with [14], which appears in these proceedings. We use the relational notation (for specifying state transition systems), the distributed systems model, and the proof rules that are presented therein. In this paper, when we say that an event has fairness, we mean "weak fairness." We also need the channel progress assumption in [14] for unreliable channels.

At any point during a construction, we have the following:

- A state transition system defined by a set of state variables $\mathbf{v} = \{v_1, v_2, \cdots\}$, a set of events $e_1, e_2, \cdots$, and an initial condition specified by the state formula *Initial*.
- A set of invariant requirements specified by state formulas $A_0, A_1, \cdots$. We use A to denote the conjunction of all the state formulas that are in the set of invariant requirements. $Initial \Rightarrow A$ holds. (We want A to be invariant.)
- A set of event requirements specified by state formulas $S_0, S_1, \cdots$. Each requirement is associated with an event. We use S_e to denote the conjunction of all the S_i's that are associated with event e. (We want S_e to hold prior to any occurrence of e.)
- A set of progress requirements $L_0, L_1, \cdots$, which are *leads-to* assertions. (To satisfy these requirements, the specification may include additional fairness assumptions for events.)
- A *Marking* consisting of (1) event requirements that are marked, (2) (A_i, e) pairs that are marked, (3) progress requirements that are marked with tags (described below), and (4) an ordering of the L_i's (to avoid circular reasoning).

We require that the Marking satisfies the following **consistency constraints**:

C1. An event requirement S_i associated with event e is marked only if $A \wedge enabled(e) \Rightarrow S_i$ holds.

C2. A pair (A_i, e) is marked only if $A \wedge S_e \wedge e \Rightarrow A_i'$ holds.

C3. A progress requirement P leads-to Q is marked with the tag *via* e_i only if the following hold:

(i) $P \wedge A \wedge A' \wedge S_{e_i} \wedge e_i \Rightarrow Q'$,

(ii) for every event $e \neq e_i$, $P \wedge A \wedge A' \wedge S_e \wedge e \Rightarrow P' \vee Q'$, and

(iii) $P \wedge A \wedge S_{e_i} \Rightarrow enabled(e_i)$.

C4. A progress requirement $L_i \equiv P$ leads-to Q is marked with the tag *via* M *using* L_j only if the following hold:

(i) for every event $e_r(m)$ that receives $m \in M$, $P \wedge A \wedge A' \wedge S_{e_r} \wedge e_r(m) \Rightarrow Q'$,

(ii) for every event $e \neq e_r(m)$, $P \wedge A \wedge A' \wedge S_e \wedge e \Rightarrow P' \vee Q'$, and

(iii) $L_j \equiv P \wedge count(M) \geq k$ leads-to $Q \vee count(M) \geq k+1$, and L_j is listed after L_i in the ordering.

C5. A progress requirement $L_i \equiv P$ leads-to Q is marked with the tag *by closure using* $L_{j_1}, \cdots, L_{j_n}$ only if P leads-to Q can be derived from A and $L_{j_1}, \cdots, L_{j_n}$ using the implication, transitivity and disjunction proof rules, and each L_{j_k} is listed after L_i in the ordering.

At any point in a construction, the Marking indicates the extent to which the requirements are satisfied by the state transition system at that point. Thus, the Marking gives us the means to backtrack to some extent in applying our heuristic.

Example on Marking: Consider a state transition system defined by integer state variables x, y both initially 0, and events $e_0 \equiv x'=x+1$ and $e_1 \equiv y'=y+1$. Let there be an invariant requirement $A_0 \equiv x=y \vee x=y+1$, a progress requirement $L_0 \equiv y \neq x \wedge x=n$ leads-to $y=n$, and an event requirement $S_0 \equiv x=y$ associated with e_0. We can mark (A_0, e_0) because $S_0 \wedge e_0 \Rightarrow A_0'$. We can mark L_0 with tag *via* e_1 because $y \neq x \wedge x=n \wedge A_0 \wedge e_1 \Rightarrow y'=n$, $y \neq x \wedge x=n \wedge S_0 \wedge e_0 \Rightarrow false$ (that is, e_0 is disabled), and $enabled(e_1)$ is *true*. (A_0, e_1) and S_0 remain to be marked.

The heuristic terminates successfully when

(a) every S_j is marked,

(b) every (A_j, e) pair is marked, and

(c) every L_j is marked.

Condition (a) implies that $A \wedge e \Rightarrow S_e$ holds for every event e, which together with condition (b) imply that $A \wedge e \Rightarrow A'$ holds. At any point in a construction, we have $Initial \Rightarrow A$. Thus, A satisfies the invariance rule. The invariance of A and condition (c) imply that each progress assertion L_j holds according to the rule indicated in its tag (*via event*, *via M*, or *closure*). There is no circular reasoning in the proof of the L_k's, because there is a serial order of the L_k's such that if L_j appears in the tag of L_i then L_j follows L_i in the ordering. Note that every event e, such that there is a progress requirement marked *via* e, must be implemented with weak fairness.

At any point in the construction, conditions (a) and (b) imply that the system satisfies the safety requirements. Condition (c) alone implies that the system satisfies the progress requirements, *assuming* that the safety requirements hold.

The construction terminates unsuccessfully whenever we have an event requirement S_i of an event e that is inconsistent with the invariant requirements or with the other event requirements of e; i.e., $S_i \Rightarrow \neg A \vee \neg S_e$ holds. The only way to mark such an S_i will be to remove the event e.

To describe the heuristic, we need to distinguish between the name of an event and the formula that specifies it. We will use e_i's to refer to event names. At different points in the construction, an event named e_i can be specified by different formulas.

Suppose a sequence of state transition systems is constructed using the heuristic. Let β and α be two successive systems in the sequence. The system refinement steps used to derive α from β

may cause some requirements that are marked for β to become unmarked for α. To minimize the unmarking of requirements, we require β and α to satisfy the following conditions:

- $v_\beta \subseteq v_\alpha$, where v_β and v_α are the state variable sets of β and α, respectively.
- $Initial_\alpha \Rightarrow Initial_\beta$, where $Initial_\alpha$ and $Initial_\beta$ are the initial conditions of α and β, respectively.
- If $\{e_1, \cdots, e_j\}$ is the set of event names of β, then $\{e_1, \cdots, e_k\}$, where $k \geq j$, is the set of event names of α. Let every event e_i of β be specified by the formula b_i. Let every event e_i of α be specified by the formula a_i. Then the following hold:
    - $A \wedge S_{e_i} \wedge a_i \Rightarrow b_i$, for $i=1, \cdots, j$.
    - $A \wedge S_{e_i} \wedge a_i \Rightarrow b_1 \vee \cdots \vee b_j \vee v_\beta{}'=v_\beta$, for $i=j+1, \cdots, k$.

If the above conditions are satisfied, we say that α is a **conditional refinement** of β, that is, a refinement of β given that the invariant and event requirements of α hold. The Marking of β is preserved for α, except in the following two cases: (1) An event requirement S_j of e_i that is marked for β becomes unmarked if and only if $A \wedge enabled(a_i) \Rightarrow S_j$ does not hold for α. (2) A progress requirement P leads-to Q that was marked *via* e_i for β becomes unmarked if and only if $P \wedge A \wedge S_{e_i} \Rightarrow enabled(a_i)$ does not hold for α.

We state a few more definitions to be used in our heuristic. Consider state formulas P and Q, and an event e. We say that P is a *weakest precondition* of Q with respect to e iff it is logically equivalent to $[\forall v': e \Rightarrow Q']$. Note that P is *false* at a state iff e is enabled at the state and its occurrence can cause Q to be falsified.[3] We say that P is a *sufficient precondition* iff it implies the weakest precondition; that is, it satisfies $[\forall v': P \wedge e \Rightarrow Q']$. We say that P is a *necessary precondition* iff it is implied by the weakest precondition; that is, it satisfies $\neg P \Rightarrow [\exists v': e \wedge \neg Q']$.

3. Sliding Window Protocol Construction: Initial Phase

Consider the distributed system topology of Figure 1. There is a producer of data blocks at entity 1, and a consumer of data blocks at entity 2. The channels may lose, duplicate, or reorder messages in transit; these are the only errors in the channels. We want data blocks to be consumed in the same order as they were produced, and within a finite time of being produced. We will construct a sliding window protocol that uses modulo-N sequence numbers to achieve this objective.

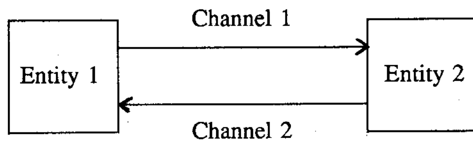

Figure 1. The network topology

[3]This corresponds to Dijkstra's weakest liberal precondition [5].

Notation: If B is a set of values, then *sequence of* B denotes the set of finite sequences whose elements are in B, and *sequence* $(0 \cdot\cdot M-1)$ *of* B denotes the set of M-length sequences whose elements are in B. For any sequence y, let $|y|$ denote the length of y, and $y(i)$ denote the ith element in y, with the 0th element at the left. Thus, $y = (y(0), \cdots, y(|y|-1))$. We use $y(i \cdot\cdot j)$ to denote $(y(i), y(i+1), \cdots, y(j))$ where $i, j < |y|$; it is null if $i > j$. We say "y prefix-of z" to mean $|y| \leq |z|$ and $y = z(0 \cdot\cdot |y|-1)$. We define the function $Tail(y, i)$ to return $y(i \cdot\cdot |y|-1)$ for any i, $0 \leq i < |y|$. Lastly, we use "wrt" as an abbreviation for "with respect to".

3.1. Initial system and requirements

The initial system and requirements specify the services to be offered to the producer and consumer. Let $DATA$ denote the set of data blocks that can be sent in this protocol. We use a Pascal-like notation to define state variables and their domains.

At entity 1, we have the following state variable and event:

produced: sequence of *DATA*. Initially null.		
$Produce(data)$	$\equiv$	$produced' = produced \ @ \ data$

At entity 2, we have the following state variable and event:

consumed: sequence of *DATA*. Initially null.		
$Consume(data)$	$\equiv$	$consumed' = consumed \ @ \ data$

The state variables *produced* and *consumed* record the sequences of data blocks produced and consumed. In the sliding window protocols to be constructed, they will be auxiliary variables. The events *Produce* and *Consume* have a parameter *data* whose domain is *DATA*.

We have one invariant requirement and two progress requirements:

A_0	$\equiv$	*consumed* prefix-of *produced*
L_0	$\equiv$	$|produced| \geq n$ leads-to $|consumed| \geq n$
L_1	$\equiv$	$|produced| \geq n$ leads-to $|produced| \geq n+1$

A_0 specifies that data blocks are consumed in the order that they are produced. It holds initially. L_0 states that if a data block is produced, then it is eventually consumed. L_1 states that at any time another data block will eventually be produced.

3.2. The sliding window mechanism

We want to refine the initial state transition system to a sliding window protocol. Let us review the basic features found in all sliding window protocols. (See Figure 2.) At any time at entity 1, the data blocks in $produced(0 \cdot\cdot a-1)$ have been sent and acknowledged, while data blocks in $produced(a \cdot\cdot s-1)$ are unacknowledged, where $|produced| = s$. At any time at entity 2, data blocks in $produced(0 \cdot\cdot r-1)$ have been received and consumed in sequence, while data blocks in

produced$(r \cdots r+RW-1)$ may have been received (perhaps out of sequence) and are temporarily buffered. The numbers r to $r+RW-1$ constitute the *receive window*; RW is its constant size.

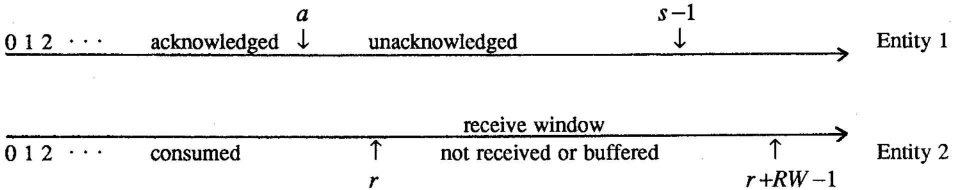

Figure 2. Relationship between a, s, r

A sliding window protocol uses modulo-N sequence numbers to identify data blocks, where $N \geq 2$. We use $\overline{n}$ to denote $n \bmod N$ for any integer value n.

Entity 1 sends *produced*(n) accompanied by sequence number $\overline{n}$. When entity 2 receives a data block with sequence number $\overline{n}$, if there is a number i in the receive window such that $\overline{i}=\overline{n}$, then the received data block is interpreted as *produced*(i). Entity 2 sends acknowledgement messages containing $\overline{n}$, where n is the current value of r. When entity 1 receives the sequence number $\overline{n}$, if there is a number i in the range $a+1$ to s such that $\overline{i}=\overline{n}$, then it is interpreted as an acknowledgement to data blocks a to $i-1$, and a is updated to i. Entity 1 increments s when a data block is produced. Entity 2 increments r when a data block is consumed.

Observe that each cyclic sequence number $\overline{n}$ corresponds to an *unbounded sequence number* n. When a cyclic sequence number is received at an entity, we require the entity to correctly interpret the value of the corresponding unbounded sequence number (which is not available in the message); that is, we require $i=n$ in the preceding paragraph.

Refinement of state transition system and requirements

We now incorporate the above protocol features into the state transition system. Let the messages sent by entity 1 be of type $(D, data, cn, n)$, where D is a constant that indicates the type of the message, *data* is a data block, *cn* is a cyclic sequence number, and n is the corresponding unbounded sequence number. Let the acknowledgement messages sent by entity 2 be of type (ACK, cn, n), where ACK is a constant that indicates the type of the message, *cn* is a cyclic sequence number, and n is the corresponding unbounded sequence number. In both message types, n is an auxiliary field that will be used to reason about correct interpretation only. Its value can never be used to update a nonauxiliary state variable. We have the following invariant requirements, each of which holds initially:

A_1	$\equiv$	$(D, data, cn, n) \in z_1 \Rightarrow data = produced(n) \wedge cn = \overline{n}$
A_2	$\equiv$	$(ACK, cn, n) \in z_2 \Rightarrow cn = \overline{n}$

At entity 1, we add the following state variables:

$s : 0 \cdots \infty$. Initially 0.
$a : 0 \cdots \infty$. Initially 0.
sendbuff: sequence of *DATA*. Initially null.

s and a are as defined above. We will ensure below that *sendbuff* always equals $produced(a \cdots s-1)$, the unacknowledged data blocks. Recall that entity 1 must retransmit these until they are acknowledged.

For brevity in specifying events, we use the notation $P \rightarrow q$ to denote an action that does q if P holds and does nothing if $\neg P$ holds. Formally, $P \rightarrow q$ means $(P \wedge q) \vee (\neg P \wedge \mathbf{x}=\mathbf{x}')$, where $\mathbf{x}$ denotes those state variables updated in q. Similarly, $[\exists i : P \rightarrow q]$ means $[\exists i : P \wedge q] \vee (\neg [\exists i : P] \wedge \mathbf{x}=\mathbf{x}')$.

At entity 1, we refine *Produce* to appropriately update *sendbuff* and s. We also add two events, one for sending data messages and one for receiving ack messages.

$Produce(data)$	$\equiv$	$produced' = produced @ data$ $\wedge\ sendbuff' = sendbuff @ data \ \wedge\ s' = s+1$
$SendD(i)$	$\equiv$	$i \in [0 \cdots s-a-1]$ $\wedge\ Send_1(D,\ sendbuff(i),\ \overline{a+i},\ a+i)$
$RecACK(cn,\ n)$	$\equiv$	$Rec_2(ACK,\ cn,\ n)$ $\wedge\ [\exists i \in [1 \cdots s-a]:\ \overline{a+i} = cn$ $\rightarrow (a' = a+i\ \wedge\ sendbuff' = Tail(sendbuff,\ i))]$

At entity 2, we add the following state variables, where *empty* is a constant not in *DATA*:

$r : 0 \cdots \infty$. Initially 0.
recbuff: sequence $(0 \cdots RW-1)$ of $DATA \cup \{empty\}$. Initially $recbuff(n) = empty$ for all n.

$r = |consumed|$ is as defined above. *recbuff* represents the buffers of the receive window. We will ensure that at any time, $recbuff(i)$ equals either *empty* or $produced(r+i)$.

At entity 2, we refine *Consume* so that it passes $recbuff(0)$ only when the latter is not empty. We also add two events, one for sending ack messages and one for receiving data messages.

$Consume(data)$	$\equiv$	$recbuff(0) \neq empty$ $\wedge\ data = recbuff(0)$ $\wedge\ recbuff' = Tail(recbuff,\ 1) @ empty \ \wedge\ r' = r+1$ $\wedge\ consumed' = consumed @ data$
$SendACK$	$\equiv$	$Send_2(ACK,\ \overline{r},\ r)$
$RecD(data,\ cn,\ n)$	$\equiv$	$Rec_1(D,\ data,\ cn,\ n)$ $\wedge\ [\exists i \in [0 \cdots RW-1]:\ \overline{r+i} = cn \rightarrow recbuff(i)' = data]$

We add the following invariant requirements; each is a desired property mentioned in the discussion above:

A_3	$\equiv$	$\lvert produced \rvert = s \ \wedge\ \lvert consumed \rvert = r$
A_4	$\equiv$	$0 \le a \le r \le s$
A_5	$\equiv$	$sendbuff = produced(a \cdots s-1)$
A_6	$\equiv$	$i \in [0 \cdots RW-1] \Rightarrow recbuff(i) = empty \vee recbuff(i) = produced(r+i)$

Marking

For the time being, we concentrate on marking the (A_i, e) pairs. We represent the Marking by a table that has a row for each A_i and a column for each e. If (A_i, e) is unmarked, its entry in the table is blank. If (A_i, e) is marked, its entry identifies a subset J of the A_j's and S_j's of e such that $J \wedge e \Rightarrow A_i'$ holds. Thus, the reader can easily check the validity of the Marking. Also, an (A_i, e) entry in the table contains *na* to indicate that e does *not affect* any of the state variables of A_i; thus $A_i \wedge e \Rightarrow A_i'$ holds trivially. We use $A_{i,j}$ to denote $A_i \wedge A_j$, and A_{i-j} to denote $A_i \wedge A_{i+1} \wedge \cdots A_j$. The *LRD* column is for the loss, reordering, and duplication events in the channels.

	Produce	*SendD*	*RecACK*	*Consume*	*SendACK*	*RecD*	*LRD*
A_0	A_0	*na*	*na*	$A_{6,3,0}$	*na*	*na*	*na*
A_1	*na*	$A_{1,5}$	*na*	*na*	*na*	A_1	A_1
A_2	*na*	*na*	A_2	*na*	A_2	*na*	A_2
A_3	A_3	*na*	*na*	A_3	*na*	*na*	*na*
A_4	A_4	*na*		$A_{6,3,4}$	*na*	*na*	*na*
A_5	$A_{5,3,4}$	*na*	A_5	*na*	*na*	*na*	*na*
A_6	$A_{6,3,4}$	*na*	*na*	A_6	*na*		*na*

The Marking can be easily checked as follows. As an example, consider the entry for $(A_4, Consume)$, which indicates that $A_{6,3,4} \wedge Consume \Rightarrow A_4'$ holds. The details are as follows: *Consume* occurs only if $recbuff(0) \ne empty$. This and A_6 imply $recbuff(0) = produced(r)$, which together with A_3 imply $r \le s-1$. This and A_4 imply $a \le r \le s-1$. *Consume* does the update $r' = r+1$ and does not affect a or s. Thus A_4' holds. In the above proof, we used A_6 first, then A_3, and then A_4. To facilitate checking of the Marking, we have indicated this in the order of the subscripts in $A_{6,3,4}$.

Observe that the only (A_i, e) pairs that are unmarked are $(A_6, RecD)$ and $(A_4, RecACK)$. We can mark $(A_6, RecD)$ if we can ensure that *RecD* correctly interprets the cyclic sequence numbers in received data messages. Similarly, we can mark $(A_4, RecACK)$ if we can ensure that *RecACK* correctly interprets the cyclic sequence numbers in received acknowledgement messages. In the next two subsections, we will generate invariant requirements on the sequence numbers that ensure correct interpretation.

3.3. Correct interpretation of data messages

In this section, we concentrate on marking $(A_6, RecD)$. Our general approach to marking an (A_i, e) pair is as follows: Obtain a weakest precondition P of A_i with respect to e; if $A \wedge S_e \Rightarrow P$

does not hold, then introduce P as a new event requirement of e; mark (A_i, e). Sometimes we simplify the expression for P to either a sufficient or a necessary precondition. In the latter case, (A_i, e) remains unmarked. Alternatively, if $Initial \Rightarrow P$ holds, we can introduce P as an invariant requirement.

The following is a weakest precondition of A_6 wrt $RecD$:

$$W \equiv Head(\mathbf{z}_1)=(D, data, cn, n) \wedge i \in [0 \cdot\cdot RW-1] \wedge \overline{r+i}=\bar{n} \Rightarrow data = produced(r+i)$$

Instead of introducing W as an event requirement, we will strengthen it to obtain a simpler sufficient precondition. From A_1, we have $cn=\bar{n}$ and $data=produced(n)$. Thus, the consequent of W is equivalent to $produced(n)=produced(r+i)$. Let us strengthen this consequent to $n=r+i$. We do not expect this to lead to unsuccessful termination. Indeed, it appears necessary in order for $produced(n)$ and $produced(r+i)$ to be arbitrary entries from $DATA$, and for the size of $DATA$ not to be limited. Next, let us weaken the antecedent of W by replacing $Head(\mathbf{z}_1)=(D, data, cn, n)$ by $(D, data, cn, n) \in \mathbf{z}_1$. In fact, this is necessary given that channel 1 can lose messages arbitrarily. Thus, we arrive at the following sufficient precondition:

$$X \equiv (D, data, cn, n) \in \mathbf{z}_1 \wedge i \in [0 \cdot\cdot RW-1] \wedge \overline{r+i}=\bar{n} \Rightarrow n=r+i$$

We decide that X will be an invariant requirement, rather than just an event requirement of $RecD$. We proceed to generate further refinements from it.

Because $produced(r)$ is the data block to be next consumed, it is reasonable to expect that $(D, data, \bar{r}, r) \in \mathbf{z}_1$ holds at any time. This would violate X with $i=N$ unless $RW \leq N$. We also know that $RW \geq 1$, otherwise entity 2 will never accept any data block and the progress requirement L_0 will never hold. Thus, we have the following condition:

$$\boxed{1 \leq RW \leq N}$$

Observe that $i \in [0 \cdot\cdot RW-1] \wedge \overline{r+i}=\bar{n}$ iff $i \in [0 \cdot\cdot RW-1] \wedge \bar{i}=\overline{n-r}$ iff $\overline{n-r} \in [0 \cdot\cdot RW-1] \wedge i=\overline{n-r}$, where we used $RW \leq N \Rightarrow i=\bar{i}$ to establish the last "iff". Thus, we can refine $RecD$ to the following, where we have also used the modulo arithmetic property $(n-r) \bmod N = (\bar{n}-r) \bmod N$:

$$\boxed{\begin{aligned} RecD(data, cn, n) \quad \equiv \quad & Rec_1(D, data, cn, n) \\ & \wedge [\overline{cn-r} \in [0 \cdot\cdot RW-1] \rightarrow recbuff(\overline{cn-r})'=data] \end{aligned}}$$

We can now refine X to the following invariant requirement:

$$Y \equiv (D, data, cn, n) \in \mathbf{z}_1 \wedge \overline{n-r} \in [0 \cdot\cdot RW-1] \Rightarrow n = r+\overline{n-r}$$

Y is satisfied nonvacuously by $n-r \in [0 \cdot\cdot RW-1]$, and satisfied vacuously by $n-r \in [RW+kN \cdot\cdot N-1+kN]$ for any integer k. We want every unbounded sequence number n in channel 1 to be in the union of these intervals. Suppose that n_1 and n_2 are in channel 1; let us assume that channel 1 may contain any n between n_1 and n_2. We expect that an n equal to r may always be in channel 1. The largest contiguous union of intervals containing r is $[r+RW-N \cdot\cdot r+N-1]$, which is the union of $[r \cdot\cdot r+RW-1]$ and $[r+RW+kN \cdot\cdot r+N-1+kN]$ for $k=0$

and −1. Thus, we strengthen Y to the following invariant requirement:

$$A_7 \equiv (D, data, cn, n) \in \mathbf{z}_1 \Rightarrow n \in [r-N+RW \cdot\cdot r+N-1]$$

We now proceed to mark $(A_7, SendD)$. A weakest precondition of A_7 wrt $SendD$ is $a \geq r-N+RW$. We will make it an invariant requirement because we want $SendD$ to be always enabled to send outstanding data. Because $r \leq s$ (and we expect $r = s$ to be possible at any time), we strengthen it to the following invariant requirement:

$$A_8 \equiv s - a \leq N - RW$$

Because A_8 only involves variables of entity 1, it can be enforced by refining *Produce* as follows:

$$\begin{aligned} Produce(data) \equiv\ & s-a \leq N-RW-1 \\ & \wedge produced' = produced \ @\ data \\ & \wedge sendbuff' = sendbuff \ @\ data \ \wedge\ s' = s+1 \end{aligned}$$

In order for *Produce* not to be permanently disabled (needed for L_1), we now require the following:

$$1 \leq RW \leq N-1$$

Observe that the upper bound in A_7's consequent is implied by $n \leq s-1$ (from $A_{1,3}$), A_4, and A_8. There is no need for A_7 to repeat this constraint. Thus, we can rewrite A_7 as follows:

$$A_7 \equiv (D, data, cn, n) \in \mathbf{z}_1 \Rightarrow n \geq r-N+RW$$

We can extend the previous Marking to the following, where * is used to indicate an old entry, and old A_i's marked wrt every event have been aggregated into one row:

	Produce	*SendD*	*RecACK*	*Consume*	*SendACK*	*RecD*	*LRD*
$A_{0-3,5}$	*	*	*	*	*	*	*
A_4	*	*		*	*	*	*
A_6	*	*	*	*	*	$A_{7,1}$	*
A_7	*na*	$A_{8,4}$	*na*		*na*	A_7	A_7
A_8	A_8	*na*	A_8	*na*	*na*	*na*	*na*

3.4. Correct interpretation of acknowledgement messages

In this section, we concentrate on marking $(A_4, RecACK)$. The treatment is similar to the case of data messages above, and we shall omit the details. We can obtain the following invariant requirements:

A_9	$\equiv$	$(ACK, cn, n) \in z_2 \Rightarrow n \le r$
A_{10}	$\equiv$	$(ACK, cn, n) \in z_2 \Rightarrow n \ge s - N + 1$

We can refine *RecACK* to the following:

$$RecACK(cn, n) \equiv Rec_2(ACK, cn, n) \wedge [\overline{cn - a} \in [1 \cdots s - a] \rightarrow (a' = a + \overline{cn - a} \wedge sendbuff' = Tail(sendbuff, \overline{cn - a}))]$$

We have the following Marking:

	Produce	*SendD*	*RecACK*	*Consume*	*SendACK*	*RecD*	*LRD*
$A_{0-3,5,6,8}$	*	*	*	*	*	*	*
A_4	*	*	A_{8-10}	*	*	*	*
A_7	*	*	*		*	*	*
A_9	*na*	*na*	A_9	A_9	A_9	*na*	A_9
A_{10}		*na*	A_{10}	*na*	$A_{10,8,4}$	*na*	A_{10}

The invariant requirements and system at this point are specified in Tables 1 and 2. Note that the only unmarked pairs are $(A_7, Consume)$ and $(A_{10}, Produce)$.

3.5. Progress requirement marking

We now try to mark L_0 and L_1. We assume that *SendD*(0) and *SendACK* have weak fairness. For the current system, we prove that L_0 holds if *Consume* has weak fairness, and that L_1 holds if *Produce* and *Consume* have weak fairness. We then show that these properties continue to hold if entity 2 sends an ack only in response to a received data message. For the progress markings in this section, we consider the L_i's to be ordered according to increasing subscripts. Hence, L_j is used in the tag of L_i only if $j > i$.

The following progress requirements imply L_0 and L_1:

L_2	$\equiv$	$s > a = n$ leads-to $a \ge n+1$
L_3	$\equiv$	$s = a = n$ leads-to $s = n+1$

We have the following Marking, where the tag also indicates the invariant requirements used to mark: L_0 by closure using L_2 and $A_{3,4}$. L_3 via *Produce*. L_1 by closure using L_2, L_3, and $A_{3,4}$ as follows: from L_2 and A_4, we have $s \ge n > a$ leads-to $s \ge a \ge n$; from this, L_3 and A_4, we get $s \ge n$ leads-to $s > n$, which is L_1 (because of A_3). At this point, only L_2 is unmarked. L_2 follows from the closure of the following progress requirements:

L_4	$\equiv$	$s>r=a=n$ leads-to $s\geq r>a=n$
L_5	$\equiv$	$s\geq r>a=n$ leads-to $a>n$

L_4 and L_5 are implied by the following progress requirements, which hold for the current system. Here, $(ACK, >n)$ denotes the message set $\{(ACK, j): j>n\}$, and (D, n) denotes the message set $\{(D, data, cn, n)\}$:

$L_6 \equiv$	$s>r=a=n$ leads-to $s\geq r>a=n \vee (recbuff(0)\neq empty \wedge s>r=a=n)$
$L_7 \equiv$	$recbuff(0)\neq empty \wedge s>r=a=n$ leads-to $s\geq r>a=n$
$L_8 \equiv$	$s>a=n \wedge count(D, n)\geq k$ leads-to $a>n \vee count(D, n)\geq k+1$
$L_9 \equiv$	$s\geq r>a=n \wedge count(ACK, >n)\geq k$ leads-to $a>n \vee count(ACK, >n)\geq k+1$

The details are summarized in the following progress Marking: L_0 by closure using L_2, $A_{3,4}$. L_1 by closure using L_2, L_3, $A_{3,4}$. L_2 by closure using L_4, L_5, A_4. L_3 via *Produce*. L_4 by closure using L_6, L_7. L_5 via $(ACK, >n)$ using L_9, $A_{4,8,10}$. L_6 via (D, n) using L_8, $A_{4,1,8}$. L_7 via *Consume* using A_4. L_8 via *SendD* (0) using A_{3-5}. L_9 via *SendACK* using A_4.

Weaker acknowledgement policy

Suppose entity 2 sends an ack message only if it has received a data message following the last ack sent. We can model this by adding a boolean variable *drecd* initially *false*, refining *RecD* by adding the conjunct *drecd'*, and refining *SendACK* to $drecd \wedge Send_2(ACK, \bar{r}, r) \wedge \neg drecd'$. The only effect of this refinement on the Marking is to unmark progress requirement L_9, which was marked *via SendACK*. However L_9 still holds because entity 1 retransmits (D, n) as long as $s\geq r>a=n$ holds. To prove this, we introduce the following progress requirements:

$L_{10} \equiv$	$drecd \wedge r>n \wedge count(ACK, >n)\geq k$ leads-to $count(ACK, >n)\geq k+1$
$L_{11} \equiv$	$s\geq r>a=n \wedge count(ACK, >n)\geq k$ leads-to $a>n \vee (drecd \wedge s\geq r>a=n \wedge count(ACK, >n)\geq k)$
$L_{12} \equiv$	$s\geq r>a=n \wedge count(ACK, >n)\geq k \wedge count(D, n)\geq l$ leads-to $a>n \vee (drecd \wedge s\geq r>a=n \wedge count(ACK, >n)\geq k) \vee count(D, n)\geq l+1$

We have a complete Marking by replacing "L_9 via *SendACK* using A_4" in the above Marking with the following: L_9 by closure using L_{10}, L_{11}. L_{10} via *SendACK*. L_{11} via (D, n) using L_{12}. L_{12} via *SendD* (0).

4. Completing the Construction for Loss-only Channels

At this point, we have obtained a system with entities as specified in Table 2. For channels that can lose, reorder, and duplicate messages, the construction is incomplete because $(A_{10}$, *Produce*) and $(A_7$, *Consume*) are not yet marked. We now show that if the channels can only lose messages, then these pairs can be marked for the current system.

We start by considering $(A_7, Consume)$. The following is a weakest precondition of A_7 wrt *Consume*:

$$(D, data, cn, n) \in z_1 \wedge recbuff(0) \neq empty \Rightarrow n \geq r + RW - N + 1$$

If instead of a single occurrence of *Consume*, we consider $k+1$ occurrences, then we obtain the following weakest precondition:

$$(D, data, cn, n) \in z_1 \wedge [\forall i \in [0 \cdot\cdot k]: recbuff(i) \neq empty] \Rightarrow n \geq r + RW - N + k + 1$$

Now if there have been no channel errors for a while, then $[\forall i \in [0 \cdot\cdot k]: recbuff(i) \neq empty]$ will hold when $recbuff(k) \neq empty$ holds. Thus, it is reasonable to strengthen the above weakest precondition to the following invariant requirement:

$$B_0 \quad \equiv \quad (D, data, cn, n) \in z_1 \wedge recbuff(k) \neq empty \Rightarrow n \geq r + k + RW - N + 1$$

The following is a weakest precondition of B_0 wrt *RecD*:

$$B_1 \quad \equiv \quad (D, d_1, cn_1, n_1) @ (D, d_2, cn_2, n_2) \text{ subseq } z_1 \Rightarrow n_2 \geq n_1 + RW - N + 1$$

We can see that B_0 is preserved by *SendD* as follows. $recbuff(k) \neq empty$ implies $s > r + k$, which together with $a \geq s - N + RW$ (A_8) implies $a > r + k - N + RW$. Thus *SendD* preserves B_0, because it sends only $produced(n)$ where $n \geq a$. The argument that *SendD* preserves B_1 is similar. $(D, d_1, cn_1, n_1) \in z_1$ implies $s > n_1$, which implies $a > n_1 - N + RW$.

We now consider marking $(A_{10}, Produce)$. Because entity 2 sends nondecreasing n and channel 2 does not reorder messages, we expect the following to be invariant:

$$B_2 \quad \equiv \quad (ACK, cn, n) \in z_2 \Rightarrow n \geq a$$

B_2 implies A_{10} because $n \geq s - N + 1$ if $n \geq a$ (from A_8). Thus marking $(B_2, Produce)$ allows us to mark $(A_{10}, Produce)$. The following is a weakest precondition of B_2 wrt *RecACK*, and is introduced as an invariant requirement:

$$B_3 \quad \equiv \quad (ACK, cn_1, n_1) @ (ACK, cn_2, n_2) \text{ subseq } z_2 \Rightarrow n_1 \leq n_2$$

At this point, we have the following complete Marking:

	Produce	*SendD*	*RecACK*	*Consume*	*SendACK*	*RecD*	*Loss*
$A_{0\text{-}6,8,9}$	*	*	*	*	*	*	*
A_7	*	*	*	B_0	*	*	*
A_{10}	B_2, A_8	*	*	*	*	*	*
B_0	*na*	$B_0, A_{8,6,3}$	*na*	B_0	*na*	$B_{0,1}$	B_0
B_1	*na*	$B_1, A_{8,1,3}$	*na*	*na*	*na*	B_1	B_1
B_2	*na*	*na*	B_3	*na*	B_2, A_4	*na*	B_2
B_3	*na*	*na*	B_3	*na*	$A_{9,4}$	*na*	B_3

5. Completing the Construction for Loss, Reordering, and Duplication Channels

For loss, reordering, and duplication channels, we resume the protocol construction from the end of Section 3, i.e., from the requirements and system shown in Tables 1 and 2, respectively. Recall that only the pairs (A_7, *Consume*) and (A_{10}, *Produce*) are unmarked.

Clearly, if the channels can reorder and duplicate arbitrarily, then A_7 and A_{10} cannot be enforced unless the channels impose an upper bound on the lifetimes of messages in transit. Therefore, we assume that a message cannot stay in channel i for longer than a specified $MAXLIFE_i$ time units. Given this, we show that A_7 and A_{10} are enforced if entity 1 produces a data block for $produced(n)$ only after (1) $MAXLIFE_1$ time units have elapsed since $produced(n-N+RW)$ was last sent, and (2) $MAXLIFE_2$ time units have elapsed since $produced(n-N+1)$ was first acknowledged. We then provide three ways to implement these two time constraints, using $2N$ timers, N timers, and 1 timer, respectively

5.1. Real-time system model

For this construction, we require a system model in which real-time constraints can be formally specified and verified. Such a real-time model has been presented in [17]. We now give a summary description of that model, adequate for our purposes here.

The system model presented is augmented with special state variables, referred to as *timers*, and with *time events* to age the timers. A timer takes values from the domain $\{OFF, 0, 1, 2, \cdots\}$. Define the function *next* on this domain by $next(OFF)=OFF$ and $next(i)=i+1$ for $i \neq OFF$. A timer can also have a maximum capacity M, for some positive integer M; in this case, $next(M)=OFF$.

There are two types of timers: *local timers* and *ideal timers*. Local timers correspond to the timers and clocks implemented within entities of a distributed system. They need not be auxiliary. For each entity, there is a *local time event* (corresponding to a clock tick) whose occurrence updates every local timer within that entity to its *next* value. No other timer in the system is affected. Thus, local timers in different entities are decoupled. We assume that the error in the ticking rate of the local time event of entity entity i is upper bounded by a specified constant ε_i; e.g., $\varepsilon_i \approx 10^{-6}$ for a crystal oscillator driven clock.

Ideal timers are auxiliary variables that record the actual time elapsed. There is an *ideal time event* whose occurrence updates every ideal timer in the system. The ideal time event is a hypothetical event that is assumed to occur at a *constant* rate. Ideal timers are used to measure the error in the rate of local time event occurrences. They are also convenient for relating elapsed times across different entities and channels.

A timer of an entity can be incremented by its time event. It can also be updated to either 0 or *OFF* by an event of that entity. Updating to the value 0 is referred to as *starting* the timer. Updating to the value *OFF* is referred to as *stopping* the timer. Thus, a timer that is started by an event occurrence measures the time elapsed since that event occurrence.

Given an ideal timer u and a local timer v of entity i, we define the predicate $started\text{-}together(u, v)$ to mean that at some instant in the past u and v were simultaneously started, and after that instant neither u nor v has been started or stopped. The maximum error in the rate of

entity i's local time event occurrences is modeled by assuming the following condition, which we shall refer to as the *accuracy axiom*:

Accuracy axiom. $started\text{-}together(u, v) \Rightarrow |u - v| \leq \max(1, \varepsilon_i u)$

An invariant requirement A_i can include *started-together* predicates. To mark (A_i, e), i.e., to derive $e \wedge A \Rightarrow A_i'$, we use the following rules. Rules (i) and (ii) are used if e is not a time event, and rule (iii) is used if e is a time event:

(i) $u'=0 \wedge v'=0$ implies $started\text{-}together(u, v)'$.

(ii) $u'=u \wedge v'=v \wedge started\text{-}together(u, v)$ implies $started\text{-}together(u, v)'$.

(iii) $u' \neq OFF \wedge v' \neq OFF \wedge started\text{-}together(u, v)$ implies $started\text{-}together(u, v)'$.

With timers and time events, time constraints between event occurrences can be specified by safety assertions. For example, let e_1 and e_2 be two events, and let v be a timer that is started by e_1 and stopped by e_2. The time constraint that e_2 *does not occur within* T time units of e_1's occurrence can be specified by the invariant requirement $enabled(e_2) \Rightarrow v \geq T$. The time constraint that e_2 *must occur within* T time units of e_1's occurrence can be specified by the invariant requirement $v \leq T$. Note that to establish the invariance of an A_i involving timers, we have to show that it is preserved by the time events also.

Specification of finite message lifetime: To every message in a channel, we add an auxiliary ideal timer field, denoted by age, that indicates the ideal time elapsed since the message was sent. The age field is started at 0 when the message is sent (this update is specified in the send primitive). The following are *assumed* to be invariant:

TA_1	$\equiv$	$(D, data, \bar{n}, n, age) \in \mathbf{z}_1 \Rightarrow MAXLIFE_1 \geq age \geq 0$
TA_2	$\equiv$	$(ACK, \bar{n}, n, age) \in \mathbf{z}_2 \Rightarrow MAXLIFE_2 \geq age \geq 0$

5.2. A time constraint that enforces A_7

In this section, we concentrate on marking $(A_7, Consume)$. We show that A_7 is enforced if entity 1 produces $produced(n)$ only after $MAXLIFE_1$ ideal time units have elapsed since $produced(n-N+RW)$ was last sent.

Due to buffered data blocks, it is always possible for successive occurrences of *Consume* to increase r so that it equals s. Unlike in the case of loss-only channels, this does not allow us to infer constraints on the sequence numbers in channel 1. Thus to enforce A_7, we require the following stronger invariant requirement to hold:

C_0	$\equiv$	$(D, data, cn, n) \in \mathbf{z}_1 \Rightarrow n \geq s-N+RW$

Taking the weakest precondition of C_0 wrt *Produce*, we get the following event requirements of *Produce*:

S_0	$\equiv$	$(D, data, cn, n) \in z_1 \Rightarrow n \geq s - N + RW + 1$

Note that this is the first precondition in this construction that we have left as an event requirement. This is because S_0 has exactly the same form as the invariant requirement C_0 from which it was derived, with N being replaced by $N-1$. Therefore, transforming S_0 into an invariant requirement would merely lead us to repeat the step with a smaller N. Repeated reductions like this would eventually lead to $N = RW$, at which point we would have a "dead" protocol because of A_8.

S_0 can be enforced by enabling *Produce* only after $MAXLIFE_1$ time units have elapsed since the last send of any data block in $produced(0 \cdot\cdot s-N+RW)$. With this motivation, we add ideal timers $t_D(n)$, $n \geq 0$, at entity 1 to record the ideal time elapsed since $produced(n)$ was last sent. We also refine *SendD* and introduce an invariant requirement as follows:

t_D: sequence $(0 \cdot\cdot \infty)$ of ideal timer. Initially $t_D(n) = OFF$ for every n.	
$SendD(i) \equiv$	$i \in [0 \cdot\cdot s-a-1] \wedge Send_1(D, sendbuff(i), \overline{a+i}, a+i) \wedge t_D(a+i)' = 0$
$C_1 \equiv$	$(D, data, cn, n, age) \in z_1 \Rightarrow age \geq t_D(n) \geq 0$

We can enforce S_0 by having $X \equiv n \in [0 \cdot\cdot s-N+RW] \Rightarrow t_D(n) > MAXLIFE_1 \vee t_D(n) = OFF$ as an event requirement of *Produce*. This would make the following invariant:

C_2	$\equiv$	$n \in [0 \cdot\cdot s-N+RW-1] \Rightarrow t_D(n) > MAXLIFE_1 \vee t_D(n) = OFF$

C_2 is preserved by *SendD* because $a > s-N+RW-1$, and by *Produce* because of X. Because C_2 is an invariant requirement, we can enforce X by enforcing the following event requirement of *Produce*:

S_1	$\equiv$	$n = s-N+RW \geq 0 \Rightarrow t_D(n) > MAXLIFE_1 \vee t_D(n) = OFF$

The above discussion is formalized in the the following Marking, which now includes event requirements, and where *Ite* denotes the *Ideal time event*:

	Produce	*SendD*	*RecACK*	*Consume*	*SendACK*	*RecD*	*LRD*	*Ite*
$A_{0-6,8,9}$	*	*	*	*	*	*	*	*na*
A_7	*	*	*	C_0, A_4	*	*	*	*na*
A_{10}		*	*	*	*	*	*	*na*
C_0	S_0	A_8	*na*	*na*	*na*	C_0	C_0	*na*
C_1	*na*	C_1	*na*	*na*	*na*	C_1	C_1	TA_1
C_2	S_1, C_2	*na*	*na*	*na*	*na*	*na*	*na*	C_2

S_0 marked using $S_1, C_{1,2}, TA_1$	S_1 not marked

To enforce S_1, it is sufficient for entity 1 to keep track of the ideal timers in $t_D(s-N+RW \cdot\cdot s-1)$. This can be done with a bounded number of local timers, each of bounded capacity.

5.3. A time constraint that enforces A_{10}

In this section, we concentrate on marking (A_{10}, *Produce*). We show that A_{10} is enforced if entity 1 produces a data block for *produced*(n) only after $MAXLIFE_2$ ideal time units have elapsed since *produced*($n-N+1$) was acknowledged.

Taking the weakest precondition of A_{10} wrt *Produce*, we get the following event requirement of *Produce* (which, as in the case of S_0, should not be transformed into an invariant requirement):

$$S_2 \quad \equiv \quad (ACK,\, cn\,,\, n) \in z_2 \Rightarrow n \geq s-N+2$$

S_2 can be enforced only by ensuring that more than $MAXLIFE_2$ time units have elapsed since $(ACK, \bar{n}, n)$ was last sent, for any $n \in [0 \cdot\cdot s-N+1]$. Unlike the previous case involving data messages, entity 1 does *not* have access to the time elapsed since $(ACK, \bar{n}, n)$ was last sent. This is because *ACK* messages are sent by entity 2 and not by entity 1. However, entity 1 can obtain a lower bound on this elapsed time because of the following considerations: $(ACK, \bar{n}, n)$ is not sent once r exceeds n; a exceeds n only after r exceeds n; a and r are nondecreasing quantities. Thus, the time elapsed since a exceeded n is a lower bound on the ages of all $(ACK, \bar{n}, n)$ in channel 2. Furthermore, this elapsed time *can* be measured by entity 1.

With this motivation, we add ideal timers $t_R(n)$, $n \geq 0$, at entity 2 to record the ideal time elapsed since r first exceeded n, and refine *Consume* appropriately (for brevity, we only indicate the addition to the previous definition given in Table 2):

t_R : sequence $(0 \cdot\cdot \infty)$ of ideal timer. Initially $t_R(n)=OFF$ for every n.
Consume(*data*) $\equiv$ <definition in Table 2> $\wedge\ t_R(r)'=0$

At entity 1, we add ideal timers $t_A(n)$, $n \geq 0$, to record the ideal time elapsed since a first exceeded n, and refine *RecACK* appropriately:

t_A : sequence $(0 \cdot\cdot \infty)$ of ideal timer. Initially $t_A(n)=OFF$ for every n.

$$\begin{aligned} RecACK(cn\,,\, n) \quad \equiv \quad & Rec_2(ACK,\, cn\,,\, n) \\ & \wedge [\overline{cn-a} \in [1 \cdot\cdot s-a] \\ & \quad \rightarrow (a' = a + \overline{cn-a} \wedge sendbuff' = Tail(sendbuff,\, \overline{cn-a}) \\ & \quad\quad \wedge [\forall i \in [a \cdot\cdot a'-1]:\ t_A(i)'=0])] \end{aligned}$$

We have the following invariant requirements:

C_3	$\equiv$	$t_R(0) \geq t_R(1) \geq \cdots \geq t_R(r-1) \geq 0 \ \wedge \ t_R(r \cdot\cdot \infty) = OFF$
C_4	$\equiv$	$(ACK, \bar{n}, n, age) \in \mathbf{z}_2 \wedge n < r \Rightarrow age \geq t_R(n) \geq 0$
C_5	$\equiv$	$t_A(0) \geq t_A(1) \geq \cdots \geq t_A(a-1) \geq 0 \ \wedge \ t_A(a \cdot\cdot \infty) = OFF$
C_6	$\equiv$	$n \in [0 \cdot\cdot a-1] \Rightarrow t_A(n) \leq t_R(n)$

From A_8, $C_{4\text{-}6}$, and TA_2, and $1 \leq RW \leq N-1$, we see that the following implies S_2:

S_3	$\equiv$	$n = s-N+1 \geq 0 \Rightarrow t_A(n) > MAXLIFE_2$

We have the following Marking. (Using $A_4{}'$ to mark some entries is acceptable because A_4 has been proven invariant; equivalently, we can replace $A_4{}'$ with its tag $A_{8\text{-}10}$):

	Produce	*SendD*	*RecACK*	*Consume*	*SendACK*	*RecD*	*LRD*	*Ite*
$A_{0\text{-}9}$, $C_{0\text{-}2}$	*	*	*	*	*	*	*	*
A_{10}	S_2	*	*	*	*	*	*	*
C_3	*na*	*na*	*na*	C_3	*na*	*na*	*na*	C_3
C_4	*na*	*na*	C_4	C_4, TA_2	C_4	*na*	C_4	C_4
C_5	*na*	*na*	C_5	*na*	*na*	*na*	*na*	C_5
C_6	*na*	*na*	C_6, $A_4{}'$, C_3	C_6, A_4	*na*	*na*	*na*	C_6

S_0 marked using S_1, $C_{2,1}$, TA_1	S_1 unmarked	S_2 marked using S_3, A_8, $C_{4\text{-}6}$, TA_2	S_3 unmarked

5.4. Protocol I: implementation with 2N timers

The only unmarked requirements are S_1 and S_3. In Table 3, we provide a system specification in which entity 1 enforces S_1 and S_3 using two circular arrays of N local timers, namely $timer_D$ and $timer_A$. (It is possible for $timer_D$ to be of size $N-RW$ and $timer_A$ to be of size $N-1$. But it involves notation for modulo $N-RW$ and $N-1$ arithmetic.)

Given an ideal timer u and a local timer v of entity 1 which are started together, from the accuracy axiom it is clear that $u > T$ holds if $v \geq 1+(1+\varepsilon_1)T$, or equivalently if v is a timer of capacity $(1+\varepsilon_1)T$ and is OFF. With this motivation, define $MLIFE_i = (1+\varepsilon_1)MAXLIFE_i$ for $i=1$ and 2.

$timer_D$ is an array $(0 \cdot\cdot N-1)$ of local timers, each of capacity $MLIFE_1$. For $n \in [\max(0, s-N+RW) \cdot\cdot s-1]$, $timer_D(\bar{n})$ tracks $t_D(n)$ up to $MLIFE_1$ local time units with an accuracy of ε_1. Thus, S_1 is enforced by including $timer_D(\overline{s-N+RW}) = OFF$, or equivalently $timer_D(\overline{s+RW}) = OFF$, in the enabling condition of *Produce*, as shown in Table 3.

$timer_A$ is an array $(0 \cdot\cdot N-1)$ of local timers, each of capacity $MLIFE_2$. For $n \in [\max(0, s-N+1) \cdot\cdot a-1]$, $timer_A(\bar{n})$ tracks $t_A(n)$ up to $MLIFE_2$ local time units with an accuracy of ε_1. Thus, S_3 is enforced by including $timer_A(\overline{s-N+1}) = OFF$, or equivalently $timer_A(\overline{s+1}) = OFF$, in the enabling condition of *Produce*, as shown in Table 3.

For brevity, we omit a formal proof that this protocol satisfies the event requirements S_1 and S_3. (It is contained in [18].)

The previous Marking of the progress requirements holds with a few minor changes. The current system is a refinement of the basic protocol, and the only event whose enabling condition has changed is *Produce*. Thus the only effect on the previous Marking is to unmark L_3, which was marked *via Produce*. The previous marking of L_2 is still valid because it did not use L_3. While L_3 does not follow immediately via *Produce*, it still holds. It is sufficient to show that the two constraints involving timers in the enabling condition of *Produce* eventually become true when $s=a$. But this holds because the time events can never deadlock [17]. Therefore, any bounded timer that is not *OFF* will eventually become *OFF*.

5.5. Protocol II: implementation with N timers

In Table 4, we provide an implementation in which both S_1 and S_3 are enforced by the N local timers in $timer_A$. Unlike in the previous implementation with $timer_D$, the enforcement of S_1 is not tight, i.e., entity 1 takes more than the minimum time to detect that S_1 holds.

Because $produced(n)$ is not sent after it is acknowledged, we have $t_D(n) \geq t_A(n)$ for all $n \in [0 \cdot\cdot a-1]$. The proof of this is trivial and is omitted. Thus, an alternative way to enforce S_1 is to enforce the following:

$$S_4 \equiv n=s-N-RW \geq 0 \Rightarrow t_A(n) > MAXLIFE_1$$

S_4 is analogous to S_3 and can be enforced by including $timer_A(\overline{s+RW}) > MLIFE_1$ in the enabling condition of *Produce*. We have to combine this with the other condition $timer_A(\overline{s+1}) > MLIFE_2$ needed to enforce S_3, as shown in Table 4. The Marking of the progress requirements is as in protocol I.

5.6. Protocol III: implementation with one timer

In Table 5, we provide an implementation that enforces S_3 and S_4 by using a single local timer. The timer, denoted by $timer_S$, imposes a minimum time interval δ between successive occurrences of *Produce*. We also require that $s-a$ does not exceed a constant, SW, which must be less than $N-RW$. The following inequalities must be satisfied by δ and SW:

$$1 \leq SW \leq N-RW-1$$
$$\delta \geq \max\left[\frac{MAXLIFE_1}{N-RW-SW}, \frac{MAXLIFE_2}{N-1-SW}\right]$$

For the typical case of $MAXLIFE_1 = MAXLIFE_2 = MAXLIFE$ the above constraint on δ simplifies to $\delta \geq \frac{MAXLIFE}{N-SW-RW}$. If in addition, N is very large compared to SW or RW (e.g. in TCP, $N=2^{32}$ while $SW, RW \leq 2^{16}$), then the bound simplifies to $\delta \geq \frac{MAXLIFE}{N}$.

Stenning [20] considered the case of $MAXLIFE_1 = MAXLIFE_2 = MAXLIFE$ and obtained the bound $N \geq SW + \max(M + RW, SW)$, where $M = \frac{MAXLIFE}{\delta}$. We get $N \geq SW + RW + M$, which is a tighter bound. Stenning's protocol also has several unnecessary requirements, as follows. Whenever the producer retransmits a data block with sequence number *i*, it also resends every outstanding data block with a sequence number larger than *i*. Whenever the consumer receives a data message, it must send an acknowledgement message.

Observe that the above time constraint on δ corresponds to specifying a maximum rate of data transmission, if we assume that *Produce* also transmits the accepted data block. (There is no loss of generality here; entity 1 need merely save in another buffer data blocks that are produced and not yet sent.) Note that if δ is sufficiently small, e.g. the hardware clock period, then there is no need for entity 1 to explicitly use a local timer. This would correspond to the situation in TCP [10] and the original Stenning's protocol [20].

6. Discussions

Our stepwise refinement heuristic is influenced by Dijkstra's work on the derivation of programs by using weakest preconditions [5], and by his development of distributed programs by incrementally adding invariants and actions to preserve the invariants [6,7,8,9].

There are differences and similarities between our approach and the approach of Chandy and Misra [3,4] to derive distributed programs by stepwise refinement. In both approaches, a distributed system is modeled by a set of state variables and events. Invariant and progress requirements are maintained throughout the construction. In the approach of Chandy and Misra, most of the effort is spent on refining the set of requirements; the distributed program is not shown until very detailed requirements have been obtained. In our approach, most of the effort is spent on refining the state transition system; the detailed requirements are derived in order to satisfy our various conditions for one state transition system to be a refinement of another state transition system.

In many of the examples of Chandy and Misra, the topology of the network of processes is refined by breaking up an event into several events, which are subsequently associated with different processes. This type of refinement step has also been used by other authors [1,2,15]. We have not found use for such a refinement step in our examples, which are from the area of communication protocols.

In summary, event requirements, a Marking, and the conditional refinement relation between specifications are unique features of our approach. Event requirements allow us to state safety requirements that cannot yet be made invariant without causing unsuccessful termination. The Marking provides a useful representation of the extent to which the requirements are satisfied by the current state transition system and fairness assumptions. The conditional refinement relation gives us some flexibility in generating new state transition systems, while keeping any decrease in the Marking to a minimum.

We find the relational notation to be very convenient for expressing event refinement, and for reasoning about the effect of an action on invariant requirements. However, our construction heuristic does not require it; events can be specified by guarded multiple-assignment statements as in [3].

Table 1: Invariant requirements for the basic protocol

Properties relating state variables at the entities

		$1 \leq RW \leq N-1$
A_0	$\equiv$	*consumed* prefix-of *produced*
A_3	$\equiv$	$\lvert produced \rvert = s \;\wedge\; \lvert consumed \rvert = r$
A_4	$\equiv$	$0 \leq a \leq r \leq s$
A_5	$\equiv$	$sendbuff = produced(a \cdot\cdot s-1)$
A_6	$\equiv$	$i \in [0 \cdot\cdot RW-1] \Rightarrow recbuff(i) = empty \vee recbuff(i) = produced(r+i)$
A_8	$\equiv$	$s - a \leq N - RW$

Properties of D messages

A_1	$\equiv$	$(D, data, cn, n) \in \mathbf{z}_1 \Rightarrow data = produced(n) \wedge cn = \bar{n}$
A_7	$\equiv$	$(D, data, cn, n) \in \mathbf{z}_1 \Rightarrow n \geq r - N + RW$

Properties of ACK messages

A_2	$\equiv$	$(ACK, cn, n) \in \mathbf{z}_2 \Rightarrow cn = \bar{n}$
A_9	$\equiv$	$(ACK, cn, n) \in \mathbf{z}_2 \Rightarrow n \leq r$
A_{10}	$\equiv$	$(ACK, cn, n) \in \mathbf{z}_2 \Rightarrow n \geq s - N + 1$

Table 2: System specification for the basic protocol

Entity 1

produced: sequence of *DATA*. Initially null.
$s : 0 \cdot\cdot \infty$. Initially 0.
$a : 0 \cdot\cdot \infty$. Initially 0.
sendbuff: sequence of *DATA*. Initially null.

$Produce(data)$	$\equiv$	$s-a \le N-RW-1$ $\wedge\, sendbuff' = sendbuff \mathbin{@} data \;\wedge\; s' = s+1$ $\wedge\, produced' = produced \mathbin{@} data$
$SendD(i)$	$\equiv$	$i \in [0 \cdot\cdot s-a-1] \;\wedge\, Send_1(D, sendbuff(i), \overline{a+i}, a+i)$
$RecACK(cn, n)$	$\equiv$	$Rec_2(ACK, cn, n)$ $\wedge\, [\overline{cn-a} \in [1 \cdot\cdot s-a]$ $\quad \rightarrow (a' = a + \overline{cn-a} \;\wedge\, sendbuff' = Tail(sendbuff, \overline{cn-a}))]$

Entity 2

consumed: sequence of *DATA*. Initially null.
$r : 0 \cdot\cdot \infty$. Initially 0.
recbuff: sequence $(0 \cdot\cdot RW-1)$ of $DATA \cup \{empty\}$. Initially $recbuff = empty$.

$Consume(data)$	$\equiv$	$recbuff(0) \ne empty$ $\wedge\, data = recbuff(0)$ $\wedge\, recbuff' = Tail(recbuff, 1) \mathbin{@} empty \;\wedge\; r' = r+1$ $\wedge\, consumed' = consumed \mathbin{@} data$
$SendACK$	$\equiv$	$Send_2(ACK, \overline{r}, r)$
$RecD(data, cn, n)$	$\equiv$	$Rec_1(D, data, cn, n)$ $\wedge\, [\overline{cn-r} \in [0 \cdot\cdot RW-1] \rightarrow recbuff(\overline{cn-r})' = data]$

Table 3: System specification for protocol I

Entity 1

produced, s, a, *sendbuff* defined as in Table 2.
t_D, t_A : sequence $(0 \cdot\cdot \infty)$ of ideal timer. Initially $t_D = t_A = OFF$.
$timer_D$: sequence $(0 \cdot\cdot N-1)$ of local timer of capacity $MLIFE_1$. Initially $timer_D = OFF$.
$timer_A$: sequence $(0 \cdot\cdot N-1)$ of local timer of capacity $MLIFE_2$. Initially $timer_A = OFF$.

Produce (*data*) ≡ $timer_D(\overline{s+RW}) = OFF \wedge timer_A(\overline{s+1}) = OFF$
∧ <definition in Table 2>

SendD (i) ≡ <definition in Table 2>
$\wedge\ timer_D(\overline{a+i})' = 0 \wedge t_D(a+i)' = 0$

RecACK (cn, n) ≡ $Rec_2(ACK, cn, n)$
$\wedge [\overline{cn-a} \in [1 \cdot\cdot s-a]$
$\rightarrow (a' = a + \overline{cn-a} \wedge sendbuff' = Tail(sendbuff, \overline{cn-a})$
$\wedge [\forall i \in [a \cdot\cdot a'-1]: t_A(i)' = timer_A(\overline{i})' = 0])]$

Entity 2

consumed, r, *recbuff* defined as in Table 2.
t_R : sequence $(0 \cdot\cdot \infty)$ of ideal timer. Initially $t_R = OFF$.

Consume (*data*) ≡ <definition in Table 2> $\wedge\ t_R(r)' = 0$

SendACK ≡ <definition in Table 2>

RecD (*data*, cn, n) ≡ <definition in Table 2>

Table 4: System specification for protocol II

Entity 1

produced, s, a, *sendbuff*, t_D, t_A defined as in Table 3.
$timer_A$: sequence $(0 \cdot\cdot N-1)$ of local timer of capacity $\max(MLIFE_1, MLIFE_2)$. Initially $timer_A = OFF$

$Produce(data) \equiv$	<definition in Table 2> $\wedge$ $timer_A(\overline{s+RW})=OFF$		if $MLIFE_1 \geq MLIFE_2$
$Produce(data) \equiv$	<definition in Table 2> $\wedge$ $timer_A(\overline{s+1})=OFF$ $\wedge$ $(timer_A(\overline{s+RW})=OFF \vee timer_A(\overline{s+RW})>MLIFE_1)$		if $MLIFE_1 < MLIFE_2$
$SendD(i)$	$\equiv$	<definition in Table 2> $\wedge$ $t_D(a+i)'=0$	
$RecACK(cn, n)$	$\equiv$	<definition in Table 3>	

Entity 2 defined as in Table 3.

Table 5: System specification for protocol III

Entity 1

produced, s, a, *sendbuff*, t_D, t_A defined as in Table 3.
$timer_S$: local timer of capacity $(1+\varepsilon_1)\delta$. Initially $timer_S = OFF$.

$Produce(data) \equiv s-a \leq SW-1 \wedge timer_S = OFF \wedge timer_S{}'=0$
$\quad \wedge\ sendbuff' = sendbuff\ @\ data \wedge s' = s+1$
$\quad \wedge\ produced' = produced\ @\ data$

$SendD(i) \equiv$ <definition in Table 4>

$RecACK(cn, n) \equiv Rec_2(ACK, cn, n)$
$\quad \wedge [\overline{cn-a} \in [1 \cdot\cdot s-a]$
$\quad\quad \rightarrow (a' = a + \overline{cn-a} \wedge sendbuff' = Tail(sendbuff, \overline{cn-a})$
$\quad\quad\quad \wedge [\forall i \in [a \cdot\cdot a'-1]: t_A(i)'=0])]$

Entity 2 defined as in Table 3.

References

[1] R.J.R. Back and R. Kurki-Suonio, "Decentralization of process nets with a centralized control," *Second ACM SIGACT-SIGCOPS Symp. on Prin. of Distr. Comput.*, Montreal, Aug. 1983, pp. 131-142.

[2] R.J.R. Back and R. Kurki-Suonio, "A case study in constructing distributed algorithms: Distributed exchange sort," *Proc. of Winter School on Theoretical Computer Science*, Lammi, Finland, Jan. 1984, Finnish Soc. of Inf. Proc. Sc., pp. 1-33.

[3] K.M. Chandy and J. Misra, "An example of stepwise refinement of distributed programs: Quiescence detection," *ACM Trans. on Prog. Lang. and Syst.*, Vol. 8, No. 3, July 1986, pp. 326-343.

[4] K.M. Chandy and J. Misra, *Parallel Program Design: A Foundation*, Addison-Wesley, Reading, MA, 1988.

[5] E.W. Dijkstra, *A Discipline of Programming*, Prentice-Hall, Englewood Cliffs, N.J., 1976.

[6] E.W. Dijkstra, L. Lamport, A.J. Martin, C.S. Scholten, "On-the-fly garbage collection: An exercise in cooperation," *Commun. ACM*, Vol. 21, No. 11, November 1978, pp. 966-975.

[7] E.W. Dijkstra, C.S. Scholten, "Termination detection for diffusing computations," *Inform. Proc. Letters*, Vol. 11, No. 1, August 1980, pp. 1-4.

[8] E.W. Dijkstra, "Derivation of a termination detection algorithm for distributed computations," Tech. Report, EWD-840.

[9] E.W. Dijkstra, "The distributed snapshot of K.M. Chandy and L. Lamport," Tech. Report, EWD-864, November 1983.

[10] *Transmission Control Protocol*, DDN Protocol Handbook: DoD Military Standard Protocols, DDN Network Information Center, SRI, MILSTD1778, Aug 1983.

[11] D.E. Knuth, "Verification of link-level protocols," *BIT*, Vol. 21, pp. 31-36, 1981.

[12] S.S. Lam and A.U. Shankar, "Protocol verification via projections," *IEEE Trans. on Software Engineering*, Vol. SE-10, No. 4, July 1984, pp. 325-342.

[13] S.S. Lam and A.U. Shankar, "Specifying implementations to satisfy interfaces: A state transition system approach," presented at the 26th Annual Lake Arrowhead Workshop on *How will we specify concurrent systems in the year 2000?*, September 1987; full version available as Technical Report TR-88-30, Department of Computer Sciences, University of Texas at Austin, August 1988 (revised June 1989).

[14] S.S. Lam and A.U. Shankar, "A relational notation for state transition systems," Technical Report TR-88-21, Department of Computer Sciences, University of Texas at Austin, May 1988 (revised August 1989); an abbreviated version appears in these proceedings of the REX Workshop on Refinement of Distributed Systems under the title "Refinement and projection of relational specifications."

[15] K. Sere, "Stepwise removal of virtual channels in distributed algorithms," *Second Int. Workshop on Dist. Alg.*, Amsterdam, 1987.

[16] A.U. Shankar and S.S. Lam, "Time-dependent communication protocols," *Tutorial: Principles of Communication and Networking Protocols*, S. S. Lam (ed.), IEEE Computer Society, 1984.

[17] A.U. Shankar and S.S. Lam, "Time-dependent distributed systems: proving safety, liveness and real-time properties," *Distributed Computing*, Vol. 2, No. 2, pp. 61-79, 1987.

[18] A.U. Shankar and S.S. Lam, "A stepwise refinement heuristic for protocol construction," Technical Report CS-TR-1812, Department of Computer Science, University of Maryland, March 1987 (revised March 1989).

[19] A.U. Shankar, "Verified data transfer protocols with variable flow control," *ACM Transactions on Computer Systems*, Vol. 7, No. 3, August 1989; an abbreviated version appears in *Proc. ACM SIGCOMM '86 Symposium*, Aug 1986, under the title "A verified sliding window protocol with variable flow control."

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