CONSTRUCTION OF SLIDING WINDOW PROTOCOLS

A. Udaya Shankar* and Simon S. Lam**

Department of Computer Sciences
University of Texas at Austin
Austin, Texas 78712

TR-86-09 March 1986

*Department of Computer Science, University of Maryland, College Park, Maryland 20742. Work supported by National Science foundation under Grant No. ECS 85-02113.

**Work supported by National Science Foundation under Grant No. ECS 83-04734.

Table of Contents

1. INTRODUCTION 2
1.1 Summary of results 3
1.2 Summary of our protocol model and construction method 5
1.3 Related work 6
1.4 Organization of this report 7
2. NECCESSARY CONDITIONS FOR CORRECT DATA TRANSFER AND RESULTING TIME BOUNDS 8
2.1 Neccessary conditions 8
2.2 Lower bound on data transfer times 9
2.3 Upper bound on data transfer times 10
3. REAL-TIME PROTOCOL SYSTEM MODEL AND SAFETY INFERENCE RULE 10
3.1 Timers and time constraints 10
3.2 Protocol system model 12
3.3 Safety inference rule 14
4. PROTOCOL CONSTRUCTION BY STEPWISE REFINEMENT 15
4.1 Initial image protocol 17
4.2 Desired safety properties 20
4.3 Refinement of A_4 and Rec_D 22
4.4 Refinement of A_5 and Rec_ACK 23
4.5 Refinement of A_{6-8} and the AcceptData event 24
4.6 An implementable local time constraint that enforces E_1 26
4.7 An implementable local time constraint that enforces E_2 28
4.8 The final protocol assuming the enforcement of E_4 and E_5 29
5. COMPLETING THE CONSTRUCTION: THREE WAYS TO ENFORCE E_4 AND E_5 31
5.1 Protocol implementation which tightly enforces E_4 and E_5 31
5.2 Protocol implementation which loosely enforces E_4 and tightly enforces E_5 33
5.3 Protocol implementation which loosely enforces E_4 and E_5 34
6. LIVENESS PROPERTIES OF THE PROTOCOLS 35
6.1 Inference rules 36
6.2 Liveness verification of the protocols 38
7. PROTOCOLS WITH REAL-TIME PROGRESS 41
7.1 Modified protocol implementation 42
7.2 Real-time progress verification 44
REFERENCES 45
Appendix A 53
Appendix B 58
Appendix C 61

Abstract

The sliding window is one of the most intricate and powerful mechanisms in communication network protocols. It is implemented with cyclic sequence numbers and provides the functions of error control, sequence control and flow control in high-level protocols (e.g., the DoD standard, TCP, and the international standard, X.25) as well as in low-level protocols (e.g., the international standard, HDLC). In this paper, we show how to construct sliding window protocols using modulo-N sequence numbers for any $N \geq 2$, and construct three sliding window protocols with different performance characteristics. In each case, the communication channels are assumed to lose, reorder and duplicate messages in transit. The first two sliding window protocols that we construct are novel. The third sliding window protocol corresponds to existing protocols such as Arpanet's TCP and the original Stenning's protocol; we obtain the minimum value of N needed in such protocols. The construction is done by stepwise refinement, using image protocols in the method of projections. The construction is driven by the safety properties desired of the protocols. These safety properties are verified during the construction. The constructed protocols are also verified to have desired liveness and real-time progress properties.

1. INTRODUCTION

A fundamental problem in communication protocols is that of achieving reliable data transfer over unreliable channels with the use of cyclic sequence numbers for identifying blocks of data [12, 7, 3, 17]. Consider the protocol system in Figure 1 consisting of protocol entities P_1 and P_2 connected by channels C_1 and C_2, where the channels can lose, reorder, and duplicate messages in transit. There is a source at P_1 that produces new data blocks, and a destination at P_2 that consumes data blocks. The protocol must ensure that data blocks are passed to the destination in a timely manner and in the same order as they were produced. The protocol uses modulo-N sequence numbers to identify both the data blocks and acknowledgements to the data blocks, where $N \geq 2$.

An informal description of the sliding window protocol follows. At any time, let data block 0, data block 1, ..., data block s-1, denote the sequence of data blocks that have been produced by the source at P_1. Of these, data blocks 0 to a-1 have been sent and acknowledged, while data blocks a to s-1 are *outstanding*, i.e, sent but unacknowledged. (See Figure 2.) P_1 can have at most SW data blocks outstanding; i.e. $s-a \leq SW$. The numbers a, a+1, ..., a+SW-1 constitute the *send window*; SW is its size. At any time at P_2, data blocks 0 to r-1 have been received and forwarded to the destination in sequence. Data block r has not yet been received, while data blocks r+1 to r+RW-1 may have been received out-of-sequence and are temporarily buffered. (See Figure 2.) The numbers r to r+RW-1 constitute the *receive window*; RW is its size.

P_1 sends data block n accompanied by sequence number n mod N. A new data block can be sent whenever $s-a < SW$ holds; when it is sent, s is incremented by 1. Outstanding data blocks can be retransmitted at any time. Such retransmissions are scheduled according to policies that are determined by the channel delay and error characteristics. The send window is not affected by either the transmission of a new data block or by the retransmissions of outstanding data blocks. When P_2 receives a data block with sequence number n mod N, if there is a number i in the receive window such that i mod N = n mod N, then the data block is considered to be data block i. If r mod N = n mod N then buffered data blocks which are in-sequence are passed to the destination and the receive window moves up. P_2 sends acknowledgement messages containing n mod N, where n is the current value of r. When P_1 receives the sequence number n mod N, if there is a number i in the range a+1 to s such that i mod N = n mod N, then data blocks a to i-1 are considered acknowledged and the bottom of the send window (indicated by a in Figure 2) is moved up to equal i.

We shall refer to the value n above as an *unbounded sequence number*, in contrast to the cyclic sequence number n mod N. For data transfer to be in the correct order, it is necessary and sufficient that when a cyclic sequence number is received at an entity, the entity must correctly interpret the value of the corresponding unbounded sequence number (which is not available in the message); i.e., i must equal n above. Because cyclic sequence numbers have a bounded domain {0,1,...,N-1}, correct interpretation occurs *if and only if* the send window, the receive window, and the unbounded sequence numbers in the channels stay within certain bounds of each other at all times. (A precise definition of correct interpretation and a derivation of these bounds may be found in Sections 4.2, 4.3 and 4.4.)

Because the channels can duplicate and reorder messages in transit, the bounds on the unbounded sequence numbers in the channels can be enforced only if messages do not stay indefinitely in the channels. Therefore, let $MaxDelay_i$ denote the maximum lifetime of messages in channel C_i, for i = 1 and 2. If this maximum message lifetime is not a physical characteristic of the channels, then it must be explicitly enforced by the channel implementations; i.e., each message must have an age field and the intermediate network nodes that implement the channels must delete messages that are too old [12, 15, 16, 17]. The particular value of the age field should obviously depend on the network delay characteristics. For example, in the Arpanet's Transmission Control Protocol (TCP) [12], each message can be assigned a maximum lifetime of up to 256 seconds.

1.1 Summary of results

We constructed a sliding window protocol by stepwise refinement. We found that to achieve correct interpretation of sequence numbers, and hence data transfer in the correct order, it is *sufficient* if the following three conditions hold before P_1 sends data block n for the first time:

(a) Data block n-N+RW has been acknowledged.

(b) At least $MaxDelay_1$ seconds have elapsed since data block n-N+RW was last sent.

(c) At least $MaxDelay_2$ seconds have elapsed since data block n-N+1 was acknowledged.

(Our construction and verification method is briefly summarized below.)

The above three conditions are also *neccessary* if P_2 is allowed to have zero reaction time. The reaction time of an entity is the time it takes to receive a message, perform associated internal processing including interaction with the source or destination, and send a response message;

We found that no other restrictions need be enforced by the protocol system to ensure correct interpretation of sequence numbers. In particular, the constructed protocol system allows P_1 to *retransmit any* outstanding data block at *any* time, and allows P_2 to send its acknowledgement sequence number message at *any* time. Thus, any retransmission policy that optimizes the protocol's performance can be used.

Three protocol implementations

Recall that SW is the size of the send window at P_1. Condition (a) is easily satisfied by having SW $\leq$ N-RW.

Conditions (b) and (c) are time constraints which P_1 must enforce through the use of timers. We found three different methods of enforcing these conditions.

The first implementation enforces conditions (b) and (c) tightly. To enforce condition (b) tightly, P_1 needs to measure the time elapsed since data block n was last sent, for s-N+RW $\leq$ n

$\leq$ s-1. This can be done with a circular array of N-RW timers, each of capacity $MaxDelay_1$. (There is no need to measure the elapsed times once they exceed $MaxDelay_1$.) To enforce condition (c) tightly, P_1 needs to measure the time elapsed since data block n was acknowledged, for $s-N+1 \leq n \leq a-1$. This can be done with a circular array of N-1 timers, each of capacity $MaxDelay_2$. Thus, with 2N-RW-1 timers, P_1 can enforce conditions (b) and (c) tightly, up to the accuracy of the timers.

The second implementation uses a single circular array of N-1 timers, each of capacity $\max(MaxDelay_1, MaxDelay_2)$. It enforces condition (b) loosely and condition (c) tightly. The timers measure the time elapsed since data block n was acknowledeged, for $s-N+1 \leq n \leq a-1$. Condition (c) can be tightly enforced, as in the first implementation. Condition (b) can also be enforced because data block n is not sent after it is acknowledged. However, the enforcement is not tight because the last send of a data block usually does not coincide with its acknowledgement.

The first two implementations may not be feasible when N is large (e.g. $N=2^{32}$ for TCP). The third implementation uses a single timer, and enforces both conditions (b) and (c) loosely. The timer is used to enforce a minimum time interval δ between sending new data blocks. In fact, this is the implementation used in TCP [12] and in the original Stenning's protocol [17] (see below). In order for this single time constraint to enforce conditions (b) and (c), it is *neccessary and sufficient* that the following δ formula holds:

$$\delta \geq \max \left(\frac{MaxDelay_1}{N\text{-}RW\text{-}SW}, \frac{MaxDelay_2}{N\text{-}RW\text{-}1} \right)$$

Note that the send window size SW must now be strictly less than N-RW.

Liveness verification

We verified that these protocols are live: i.e., once data block n becomes outstanding, then it is acknowledged within a finite time, provided that P_1 does not continuosly avoid retransmitting data block n, C_1 does not continuously lose messages containing data block n, and C_2 does not continuously lose acknowledgements of data block n.

Real-time progress verification

A real-time progress property certifying progress within a bounded response time T is more realistic than the liveness property described above. This is because, in practice, if progress is not achieved within a bounded time T, then P_1 aborts or resets the connection with P_2.

Let $Delay_i$ ($\leq MaxDelay_i$) be the delay that a message is *expected* to encounter in channel C_i. The message is said to be *overdelayed* if it is not received within $Delay_i$ time of its send. Define the *round trip delay* as the sum of $Delay_1$, $Delay_2$, and the maximum reaction time of P_2. We verified the following *worst-case progress* property: once data block n becomes the next data block to be acknowledged, then it is acknowledged within time T = (m $\times$ round trip delay), provided P_1 retransmits data block n at least once every round trip delay seconds, and the channels do not overdelay a particular message more than m times for some $m \geq 1$.

Performance of the protocol implementations

We examined the performance of the three protocol implementations under the assumption that the entities have zero reaction times. Given this assumption, the performance is governed entirely by the following factors: the maximum message lifetimes, via conditions (b) and (c); the channel errors and delays experienced by the messages; the values of SW and RW; and the retransmission policies utilized by P_1 and P_2.

Let the *acknowledgement time* of data block n denote the elapsed time from the beginning (i.e. first transmission of data block 0) to the moment when data block n is acknowledged at P_1. Acknowledgement times are our measure of performance. We obtain tight lower and upper bounds for the acknowledgement times in each of the three protocol implementations. The lower bounds are realized in the *best-case* scenario of negligible channel delays. The upper bounds are realized under the worst-case progress assumption described above.

The best-case performance of the first protocol implementation is optimal because it enforces the conditions (b) and (c) tightly, and because these conditions are neccessary. Because there is no restriction on the retransmission policies at P_1 and P_2, this protocol implementation can also achieve optimal performance in the average case.

The second protocol implementation achieves the same performance as the first in the best and worst cases, but not in the average. Recall that this protocol implementation enforces condition (b) loosely and condition (c) tightly.

The third protocol implementation, the one used in TCP and in the original Stenning's protocol, is worse than the first two protocol implementations in the best case, worst case, and average case. Recall that it uses only 1 timer and enforces conditions (b) and (c) loosely.

1.2 Summary of our protocol model and construction method

We use the distributed system model and inference rules developed in [14, 15]. Each process, i.e. entity or channel, in the protocol system has a set of state variables and a set of events. Each event is specified by a *predicate* that relates the values of the system state variables immediately before the event occurrence to their values immediately after the event occurrence. The predicate embodies specifications of both the event's enabling condition and action. The model includes timer variables to record the elapse of time, and time events to age the timer variables. The time events of different processes are uncoupled and their rates of occurence lie within specified error bounds. The combination of the event-driven structure, the state variables, and the predicate specification of events gives rise to fairly simple inference rules for safety and liveness. Real-time properties can be stated and verified as safety properties.

Construction method

The protocol system is constructed in successive steps. At each step of the construction, we have a partially completed version of the final protocol system, which is referred to as an *image protocol system* [9, 13], and a set of *desired* safety properties which are only partially satisfied by the image protocol system. More precisely, each desired safety property is preserved by only a subset of the events of the image protocol at that step.

We start the construction with a very simple image protocol that merely formalizes the English description given in paragraphs 2 and 3 of the introduction, and a set of desired safety properties that define correct data transfer and correct interpretation of received sequence numbers.

At each step of the construction (see Section 4 for details), we examine a desired safety property with respect to an event that does not preserve the property, and modify the events and the set of safety properties. Each new event is obtained by a *refinement* of some existing event. This ensures that, for any event, any safety property which was preserved by the event prior to the modification continues to be preserved by that event. The construction terminates when all the desired safety properties are verified, i.e. preserved by all the events. *Events, rather than processes, are the units of composition in our construction.*

1.3 Related work

To our knowledge, this is the first verified construction of sliding window protocols using modulo-N sequence numbers where N is arbitrary. It demonstrates that the use of cyclic sequence numbers and measures of real time actually *simplify* the understanding, verification, and ultimately the construction of communication protocols.

Our first two protocol implementations, using 2N-RW-1 timers and N-1 timers respectively, appear to be unique. In [14, 15], we presented these protocols for the special case of N=2, and verified their safety, liveness and real-time properties.

Our third protocol implementation corresponds closely to the original Stenning's protocol in [17], and to the Arpanet's TCP [12]. All three protocols utilize modulo-N sequence numbers, a receive window of size RW, and a send window of size SW, where $1 \leq SW \leq N-RW-1$. Stenning's protocol and TCP do not explicitly use a timer to enforce a minimum time interval δ between successive sends of new data blocks. However, they assume that such a δ exists due to hardware restrictions at the entities. We will now compare the values of N used in Stenning's protocol and in TCP with the optimal values given by the δ formula.

Comparison with the original Stenning's protocol

The formal specification of Stenning's protocol utilizes unbounded sequence numbers and requires the following transmission policies: whenever P_1 retransmits a data block, it also retransmits all succeeding outstanding data blocks in sequence; whenever P_2 receives a data block, it must send an acknowledgement before receiving the next data block. Using this specification, Stenning verified certain safety properties of the unbounded sequence numbers sent into the channels. From these safety properties, he informally demonstrated that if channel C_i retains only the last M messages sent by P_i, then the unbounded sequence numbers can be replaced by cyclic sequence numbers provided the cycle period $N \geq SW+\max(M+RW, SW)$.

We can relate Stenning's protocol to our protocol by setting $MaxDelay_1 = MaxDelay_2 = M \times \delta$. Substituting this in the δ formula, we see that it is sufficient, and neccessary, to use $N \geq SW+RW+M$. Furthermore, unlike in Stenning's protocol, any retransmission policy at P_1 and P_2 can be used in our protocol.

Comparison with TCP

TCP utilizes a 32-bit sequence number for every data byte. Each packet has an age field which can be set to any value up to 256 seconds. If we let $N = 2^{32}$, $MaxDelay_1 = MaxDelay_2 = 256$ seconds, and $RW+SW < 2^{24}$ (which is a safe assumption), we obtain $\delta \geq 2^{-4}$ microseconds. In other words, if P_1 is sending packets containing 1 Kbyte data segments, then successive packets should be sent separated by at least 64 microseconds. Clearly, there is no danger of this condition being violated by existing Arpanet hardware. However, we note that the TCP manual [12] does not impose any such lower bound on the time between sending new data blocks. (The manual does impose a lower bound of 1 microsecond in another context, namely, choosing an initial sequence number in connection establishment.)

Comparison with other protocol verifications

Later investigations [6, 10] of Stenning's protocol have considered safety and liveness properties, but not real-time progress properties. However, they have invariably considered unbounded sequence numbers, as well as the simplified situation where SW=RW=1. The safety property proofs for this relatively simple case do *not* generalize to arbitrary values of N, SW and RW. However, it is interesting to note that the liveness proofs and the real-time progress proofs for the situation SW=RW=1 [15] *do* extend to the case of arbitrary N, SW and RW examined in this paper. Note that when SW=1, the options for retransmission policies are quite limited because at most one data block can be outstanding.

A version of TCP with unbounded sequence numbers and arbitrary RW has been verified using a theorem-prover in [5].

Other related work involves the HDLC protocol [7], which uses modulo-N sequence numbers (N=8 or 128), and the alternating-bit protocol [1], which uses modulo-2 sequence numbers. Versions of HDLC using cyclic sequence numbers have been modeled and verified in [13, 2]. There are numerous verifications of the alternating-bit protocol, among them [1, 6]. We point out that both the operation and logical properties of the sliding window protocols described herein are fundamentally different from the HDLC and alternating-bit protocols, the latter being data link protocols which assume that channels may lose messages in transit, but may not reorder or duplicate them.

1.4 Organization of this report

In Section 2, we show that the three conditions specified above are neccessary for correct data transfer if P_2 has zero reaction time. We also obtain lower and upper bounds on the times at which data blocks are acknowledged, assuming that these three conditions are tightly enforced. In Section 3, we briefly outline our real-time protocol system model and the inference rule for establishing safety properties. In Section 4, we constructively derive a protocol system which is complete except for the implementation of the two time constraints represented by conditions (b) and (c) above. The construction includes a verification that this protocol system achieves correct data transfer, provided that the two time constraints are enforced. In Section 5, we complete the protocol construction by providing three protocol system implementations. The implementations differ in how they enforce the two time constraints. In Section 6, we outline the

inference rule for proving liveness properties, and verify the liveness of the protocol implementations. In Section 7, we verify the worst-case progress assumption for the protocols.

2. NECCESSARY CONDITIONS FOR CORRECT DATA TRANSFER AND RESULTING TIME BOUNDS

We show the neccessity of the conditions (a), (b) and (c) for correct data transfer. We obtain lower and upper bounds on the times at which data blocks are acknowledged, provided that P_1 tightly enforces these conditions.

2.1 Neccessary conditions

For n $\geq$ 0, let f_n denote the time when data block n is first sent by P_1. Let l_n denote the time when data block n is last sent by P_1. Let a_n denote the time when data block n is acknowleged at P_1.

Because data block n will not be sent once it is acknowledged, we have the following:

R_0 For all n: $a_n \geq l_n \geq f_n$

Because acknowledgements are cumulative, we have the following:

R_1 For all n: $a_n \leq a_{n+1}$

Whenever data blocks m, m+1, ..., n are outstanding at P_1, it is possible for the receive window at P_2 to correspond to the data blocks j, j+1, ..., j+RW-1, where j can be any value from m, m+1, ..., n+1. This can happen for example, if data blocks m to j-1 have been received in order at P_2, and the rest have not been received.

If $f_n \leq l_{n\text{-}N+RW}$ + MaxDelay_1 for some n, then the following erroneous behavior can occur starting at f_n: P_2's receive window corresponds to data blocks n to n+RW-1; P_2 receives data block n immediately, passes it to the destination, and moves its receive window to data blocks n+1 to n+RW; P_2 then receives an old duplicate data block n-N+RW, which may still be in channel C_1, and misinterprets it as data block n+RW. Thus, it is neccessary that the following hold:

R_2 For all n: $f_n > l_{n\text{-}N+RW} + \text{MaxDelay}_1$

where l_i is treated as -∞ for i<0. R_2 corresponds to condition (b) in Section 1.1.

If $f_n \leq a_{n\text{-}N+1} + \text{MaxDelay}_2$ for some n, then the following erroneous behavior can occur: Just prior to $a_{n\text{-}N+1}$, P_2's receive window was data blocks n-N+1 to n-N+RW, and it had sent an acknowledgement message containing (n-N+1) mod N. Immediately afterwards, P_2 received data block n-N+1, and sent the appropriate acknowledgement message (containing (n-N+2) mod N) which was received immediately at P_1 at time $a_{n\text{-}N+1}$. The old acknowledgement message

containing (n-N+1) mod N can still be in channel C_2 just after time instant f_n. At this moment, P_1 receives it and misinterprets it as an acknowledgement to data block n. Thus, it is neccessary that the following hold:

R_3 For all n: $f_n > a_{n-N+1} + \text{MaxDelay}_2$

R_3 corresponds to condition (c) in Section 1.1.

P_1 must retransmit outstanding data blocks until they are acknowledged. This is because it is always possible that an outstanding data block is neither in C_1 nor has been received at P_2. Therefore, to enforce R_2, it is neccessary that P_1 send data block n only after data block n-N+RW is acknowledged. This corresponds to condition (a) in Section 1.1. Because SW is the send window size at P_1, we can state this condition as the following:

R_4 For all n: $f_n > a_{n-SW}$

where SW $\leq$ N-RW. We will see that the value of SW has no effect on either the lower or upper bounds of the performance. SW does affect the average performance.

In this section, we assume that P_1 has negligible reaction time, i.e., enforces R_2, R_3 and R_4 tightly. Thus, we have the following:

R_5 For all n: $f_n = \max(a_{n-SW}, l_{n-N+RW}+\text{MaxDelay}_1, a_{n-N+1}+\text{MaxDelay}_2)$

2.2 Lower bound on data transfer times

The best-case, or minimum-valued solution to R_5 is obtained when a data block is sent exactly once and acknowledged immediately. Assuming negligible channel delays and entity reaction times, we have R_0 holding with equality. Combining this with R_5, we see that the lower bound is the solution to the following equation (notice that R_4 is automatically enforced because of R_1):

R_6 $a_n = \max(a_{n-N+RW}+\text{MaxDelay}_1, a_{n-N+1}+\text{MaxDelay}_2)$

We have the following theorem:

Theorem 1. The best-case $\{a_n\}$ satisfies the following:

(a) $a_n \geq \max(\lfloor \frac{n}{N-RW} \rfloor \times \text{MaxDelay}_1, \lfloor \frac{n}{N-1} \rfloor \times \text{MaxDelay}_2)$

(b) $a_n = \lfloor \frac{n}{N-RW} \rfloor \times \text{MaxDelay}_1$ if $\lfloor \frac{N-1}{N-RW} \rfloor \times \text{MaxDelay}_1 \geq \text{MaxDelay}_2$

Theorem 1 can be easily proved by substituting the expressions for $\{a_n\}$ in R_6. Notice that case (b) in Theorem 1 includes the typical situation of $\text{MaxDelay}_1 = \text{MaxDelay}_2$.

2.3 Upper bound on data transfer times

We shall obtain upper bounds on $\{a_n\}$ under the assumptions of R_5 and the worst-case progress assumption. The latter states that the time interval during which data block n is both outstanding and the next to be acknowledged is upperbounded by a value of T seconds. It can be formally stated as follows:

$R_7 \quad a_n \leq \max(f_n, a_{n-1}) + T$

We choose this particular inequality for the worst-case progress assumption, rather than say $a_n - f_n \leq T$, for two reasons. First, it is relatively easy to design protocols that enforce it, provided the channels do not consistently perform badly (see Section 7). Second, it is a realistic assumption in that most protocols will reset if no progress is achieved within the specified time T.

The worst-case $\{a_n\}$ is the solution to the following equation, which is obtained by substituting R_7 with equality into R_5:

$R_8 \quad a_n = \max(a_{n-1}, a_{n-N+RW} + \text{MaxDelay}_1, a_{n-N+1} + \text{MaxDelay}_2) + T$

Notice that a_{n-SW} drops out of R_8 because of R_1.

Theorem 2. Let $\Delta_1 = \text{MaxDelay}_1 - (N-RW-1) \times T$, and $\Delta_2 = \text{MaxDelay}_2 - (N-2) \times T$.

(a) $a_n = (n+1) \times T$ if $\Delta_1, \Delta_2 \leq 0$

(b) $a_n \leq (n+1) \times T + \lfloor \frac{n}{N-RW} \rfloor \times (\max(\text{MaxDelay}_1, \text{MaxDelay}_2) - (N-RW-1) \times T)$ otherwise

(c) $a_n = (n+1) \times T + \lfloor \frac{n}{N-RW} \rfloor \times \Delta_1$ if $\lfloor \frac{N-RW}{N-1} \rfloor \times \Delta_1 \geq \Delta_2$

Theorem 2 can be easily proved by substituting the expressions for $\{a_n\}$ in R_8. Notice that case (c) includes the typical situation of $\text{MaxDelay}_1 = \text{MaxDelay}_2$.

3. REAL-TIME PROTOCOL SYSTEM MODEL AND SAFETY INFERENCE RULE

In Section 3.1, we outline our modeling of timers and time constraints. In Section 3.2, we present the event-driven model for the protocol system. In Section 3.3, we present the inference rules for verifying safety properties.

3.1 Timers and time constraints

We use the term *local timers* to refer to the timers that are implemented within individual processes of a distributed system. In our model, a local timer is a discrete-valued state variable that can take values from the domain {Off,0,1,2,...}. For this domain, define the successor function *next* as follows: *next*(Off)=Off and *next*(i)=i+1 for $i \neq$Off. To model timers with limited counting capacity, we also allow a local timer to have a domain {Off,0,1,...,M} where M is some

positive integer. In this case, *next*(M)=Off. When a timer is *aged*, if the original value is t then its new value is *next*(t).

For each process, there is a *local time event* (corresponding to a clock tick) whose occurrence ages all local timers within that process. Since no other timer is affected, local timers in this process are effectively decoupled from local timers in other processes of the distributed system. In addition to being aged by its time event, a local timer can be *reset* to some value by an event (other than the time event) of that process, thereby measuring the time elapsed (in number of occurrences of its local time event) following the event occurrence.

In order to keep the aging rate of local timers in different processes within specified error bounds, we include in our model a hypothetical time event, referred to as the *ideal time event*, that is assumed to occur at an absolutely constant rate. For process i, let η_i denote the number of occurrences of its local time event since system initialization, and let ϵ_i denote the maximum error in the tick rate (e.g., for a crystal oscillator, $\epsilon_i \approx 10^{-6}$). Let η denote the number of occurrences of the ideal time event since initialization. The η's are auxiliary state variables that are not implemented, and can never be reset by any event. Neither the local time event for process i nor the ideal time event is allowed to occur if such an occurrence will violate the following *accuracy axiom* of the local time event of process i

$$\text{AccuracyAxiom}_i(\eta_i,\eta)\text{: For any earlier instant } a, \\ \mid (\eta_i - \eta_i(a)) - (\eta - \eta(a)) \mid \leq \max(1,\ \epsilon_i(\eta - \eta(a))).$$

where $\eta(a)$ refers to the value of η at instant a, and η refers to the current value of the state variable η. This is a discrete version of the condition $\mid 1 - \frac{d\eta_i}{d\eta} \mid \leq \epsilon_i$, which is used to characterize the accuracy of continuous timers [8].

Implementable time constraints

Implementable time constraints are time constraints that are realizable by individual processes without any cooperation from the rest of the distributed system. They are guaranteed by the implementations of individual processes, and are not properties that have to be verified by analyzing the interaction of processes.

Recall that timer variables can measure the time elapsed since a system event occurrence. Implementable time constraints of the form "event e *will occur only if* elapsed times satisfy certain bounds" are modeled by including timer variables in the enabling condition of event e. Implementable time constraints of the form "event e *must occur within* certain elapsed times" are modeled by imposing conditions, referred to as *timer axioms*, on the time event of the process.

In [15], we have provided a formal definition of implementable time constraints, and shown that such time constraints will never cause the time events to deadlock. Observe that the time events are completely defined by the ideal and local timers, the timer axioms, and the error rates of the local time events. Time events are not implemented.

Derived time constraints and ideal timers

Derived time constraints are time constraints that hold for the distributed system as a result of individual processes enforcing implementable time constraints. They are not guaranteed by the implementation but must be verified for the distributed system.

To facilitate such verification, we allow timers that are driven by the ideal time event. Such timers are referred to as *ideal timers*. Ideal timers are *not* available to the implementation. Rather they are auxiliary variables used to measure the actual, or ideal, time elapsed between event occurrences. Because ideal timers throughout the system are coupled, it is more convenient to use ideal timers, rather than local timers, in verifying the relationships between implementable time constraints (enforced within processes) and the resulting system-wide time constraints. Once the desired implementable time constraints have been determined in terms of ideal timers, we can use local timers to track the values of the ideal timers within the specified error rates, and thereby implement the time constraints (see examples in Sections 4 and 5).

Given an ideal timer u and a local timer v, we say ((u,v) *started at* a) to mean that at some instant in the past u and v were simultaneously reset to the value a, and after that instant there has been no reset to either u or v. If ϵ_i is the error rate of the local time event that drives timer v, then the following *started-at property* clearly holds: ((u,v) started at a) $\Rightarrow$ $|u-v| \leq \max(1, \epsilon_i(u-a))$. Recall that a timer v with domain $\{Off,0,1,...,M\}$ is set to Off if its value was M before the occurrence of the time event. For the purposes of the started-at property, this action is treated as a reset, and not as aging.

3.2 Protocol system model

We model the distributed system of Figure 1 by a set of state variables whose values indicate the system state, a set of events that cause changes to the state variable values, and a set of initial conditions on the state variables.

State Variables

For each protocol entity P_i, let $\mathbf{v}_i$ be the set of state variables of P_i. $\mathbf{v}_i$ can have auxiliary variables needed for verification only, and timers needed to model time constraints. All local timers in $\mathbf{v}_i$ (if any) are driven by a local time event of count η_i and maximum error rate ϵ_i.

With each message in channel C_i, we associate a timer *age* that indicates the age of the message (time spent in the channel). For notational convenience, we assume that age is an ideal timer. Let the state variable $\mathbf{z}_i$ denote the sequence of <message,age> value pairs in C_i. Initially, $\mathbf{z}_i$ is the null sequence. The maximum message lifetime MaxDelay_i constraint is modeled by the following timer axiom:

$$\text{TimerAxiom}_i(\mathbf{z}_i) : \text{For every} <\text{message,age}> \text{in } \mathbf{z}_i\text{: } 0 \leq \text{age} \leq \text{MaxDelay}_i$$

The *system state vector* is defined by $\mathbf{v} = (\mathbf{v}_1, \mathbf{v}_2, \mathbf{z}_1, \mathbf{z}_2)$. The allowed initial values of $\mathbf{v}$ is specified by the initial conditions on the state variables.

Specifying events by predicates

Each process in the distributed system has a set of events. The events of entity P_i model message receptions, message sends, and associated state changes to $\mathbf{v}_i$. The events of channel C_i model channel errors such as loss, duplication, and reordering of messages in transit. If there are timers in the system model, then we also have time events.

Each event e can occur only when the state vector $\mathbf{v}$ has certain values. Its occurrence causes the state vector $\mathbf{v}$ to assume a new value. Instead of using algorithmic code, we specify the event by a predicate that relates the values of the state vector before and after the event occurrence. Specifically, an event is specified by a predicate whose free variables come from $\mathbf{v}$ and $\mathbf{v}''$, where $\mathbf{v}$ is understood to denote the value of the state vector immediately before the event occurrence and $\mathbf{v}''$ is understood to denote the value of the state vector immediately after the event occurrence. Such predicates are referred to as *event predicates*. Notice that we use the term "variable" in the mathematical sense, and the term "state variable" in the programming language sense, i.e., to denote both a location where a value may be stored, as well as the stored value.

For example, let the state vector $\mathbf{v}=(v_1, v_2)$, where v_1 and v_2 are integer-valued state variables. An event that increments the value of v_1 by the value of v_2 provided that v_1 is less than 5 is specified by the event predicate ($v_1<5$ **and** $v_1''=v_1+v_2$ **and** $v_2''=v_2$). (We use **and** and **or** to denote logical conjunction and disjunction respectively.) An event that assigns to v_1 either the value 1 or the value 2, where the choice is made nondeterministically, is specified by the predicate (($v_1''=1$ **or** $v_1''=2$) **and** $v_2''=v_2$). For compactness in specifying events, we adopt the convention that any variable v'' in $\mathbf{v}''$ that does not occur in an event predicate is not affected by the event occurrence; i.e., the conjunct $v''=v$ is implicit in the event predicate. Thus, the above two examples can be written as ($v_1<5$ **and** $v_1''=v_1+v_2$), and as ($v_1''=1$ **or** $v_1''=2$), respectively.

Given a predicate p with free variables from $\mathbf{v}$ and $\mathbf{v}''$, the notation $e(\mathbf{v}; \mathbf{v}'')\equiv p$ declares that $e(\mathbf{v}; \mathbf{v}'')$ refers to p; for example, $e(v_1,v_2; v_1'',v_2'') \equiv$ ($v_1<5$ **and** $v_1''=v_1+v_2$). The expression to the right of "$\equiv$" is referred to as the *body* of $e(\mathbf{v}; \mathbf{v}'')$. The expression to the left of the "$\equiv$" is referred to as the *header*. For any given value of $\mathbf{v}$ and $\mathbf{v}''$, we shall also use $e(\mathbf{v}; \mathbf{v}'')$ to denote the value that the predicate evaluates to. Thus, in the above example, e(1,2;3,2) is True while both e(1,2;4,2) and e(6,1;7,1) are False.

Channel events and primitives

For each channel C_i, the channel behavior (loss, reordering, etc.) can be specified by an event predicate to be called ChannelError($\mathbf{z}_i$; $\mathbf{z}_i''$). In addition, each channel has *send* and *receive primitives*. The send primitive for channel C_i is defined by $\mathrm{Send}_i(\mathbf{z}_i,m;\mathbf{z}_i'') \equiv (\mathbf{z}_i=(\mathbf{z}_i,<m,0>))$, i.e., append message m with an age of 0 to the tail of $\mathbf{z}_i$. (We will use a comma as the concatenation operator, and use parentheses to resolve ambiguities.) The receive primitive for channel C_i is defined by $\mathrm{Rec}_i(\mathbf{z}_i;m,\mathbf{z}_i'') \equiv$ (for some t)$[(<m,t>,\mathbf{z}_i'')=\mathbf{z}_i]$, i.e., remove the message at the head of $\mathbf{z}_i$ if $\mathbf{z}_i\neq$null, and assign it to m irrespective of its age. (We

use "*for all* x *in* X" and "*for some* x *in* X" to denote universal quantification and existential quantification, respectively, of variable x over domain X. The scope of the quantification is enclosed by square brackets.)

Entity events

The entity events of P_i can involve, in addition to entity state vector $\mathbf{v}_i$, the channel state variables $\mathbf{z}_1$ and $\mathbf{z}_2$, but only via the send and receive primitives. When these primitives are used in the predicate bodies of entity events, the formal message parameter m is replaced by the actual message sent or received. For example, an event e_1 of P_1 that sends the value of x into C_1 if y<5, where x and y are integer state variables in $\mathbf{v}_1$, is specified by

$$e_1(\mathbf{v}_1,\mathbf{z}_1;\mathbf{v}_1'',\mathbf{z}_1'') \equiv (y<5 \textbf{ and } \mathrm{Send}_1((x),\ \mathbf{z}_1;\mathbf{z}_1''))$$

An event e_2 that receives a message consisting of an integer from C_2 and accumulates it in x, is specified by

$$e_2(\mathbf{v}_1,\mathbf{z}_2;\mathbf{v}_1'',\mathbf{z}_1'') \equiv (\text{for some integer } n)[\mathrm{Rec}_2(\mathbf{z}_2;\ (n),\mathbf{z}_2'') \textbf{ and } x''=x+n]$$

The time events are similarly specified in a straightforward manner (see the example in Sections 4 and 5).

3.3 Safety inference rule

The set of all possible value assignments to the system state variables defines the state space of the system. Each event specifies a set of transitions between system states, each transition corresponding to a pair of system states that satisfies the event predicate. Let the predicate Initial(**v**) specify the initial conditions of the system; i.e. any initial system state satisfies Initial(**v**). A system state that can be reached from an initial system state via a sequence of event transitions is referred to as a *reachable system state.* Any realization of system behavior is represented by some path of event transitions from an initial state.

A safety property of the distributed system states relationships between values of the system state variables. It can be represented by a predicate in the variables of the global state vector **v**. An example of a safety property involving two integer state variables x and y is $(x \leq y \leq x + 1)$. A safety property $A_0(\mathbf{v})$ holds for the system if it holds at every reachable state. Such a property is said to be *invariant.* We now present the inference rule for proving invariance. (By convention, variables in the predicates are universally quantified over their domains unless otherwise indicated.)

Inference Rule for Safety. If I(**v**) is invariant and A(**v**) satisfies

(i) $\mathrm{Initial}(\mathbf{v}) \Rightarrow A(\mathbf{v})$

(ii) for every event e: $(I(\mathbf{v}) \textbf{ and } A(\mathbf{v}) \textbf{ and } e(\mathbf{v};\mathbf{v}'')) \Rightarrow A(\mathbf{v}'')$

(iii) $A(\mathbf{v}) \Rightarrow A_0(\mathbf{v})$

then we can infer that $A_0(\mathbf{v})$ is invariant.

The validity of the rule is obvious. Part (i) ensures that A(**v**) holds initially, while part (ii) ensures that A(**v**) is preserved by any event occurrence. Thus A(**v**) is invariant. Therefore, by part (iii), A_0(**v**) is invariant.

A_0(**v**) represents a desired safety property. I(**v**) can be any safety property whose invariance has already been verified. In particular, any timer or accuracy axiom can be a conjunct of I(**v**). Because I(**v**) is given to be invariant, we can replace I(**v**) by (I(**v**) **and** I(**v**")) in part (ii) of the above inference rule; this strengthening of the left hand side often helps in deriving A(**v**"). Generating A(**v**) from A_0(**v**), I(**v**), and the system specifications is a nontrivial task analogous to generating loop invariants in program verification. In Section 4, we construct both A(**v**) and the system specifications from A_0(**v**).

For brevity, we shall refer to a predicate A(**v**) as simply A, and use A" to refer to A(**v**"). Also, we shall refer to an event e(**v**;**v**") as simply e. Thus, the predicate in part (ii) is stated as (A **and** e)⇒A".

Safety assertions with started-at statements

In order to verify relationships between ideal timers and local timers, we will allow safety assertions to contain started-at statements; e.g., x>y ⇒ ((u,v) started at 0). We next present rules to be used when applying the safety inference rule to such assertions (below, u is an ideal timer and v is a local timer).

In part (i) of the safety inference rule, we can use the rule
(u=a **and** v=a) ⇒ ((u,v) started at a)

In part (ii) of the safety inference rule, if the event e being considered is a system event, then the following two rules apply:
(u"=a **and** v"=a) ⇒ ((u",v") started at a), and
(u"=u **and** v"=v **and** ((u,v) started at a)) ⇒ ((u",v") started at a)

The started-at property is preserved by a time event occurrence, unless one of the timers is a bounded capacity timer which is aged beyond its capacity to Off by the time event occurrence. Thus, the following rule applies in part (ii) of the safety inference rule, if the event e being considered is a time event:
((u≠Off ⇒ *next*(u)≠Off) **and** (v≠Off ⇒ *next*(v)≠Off) **and** ((u,v) started at a))
⇒ ((u",v") started at a)

4. PROTOCOL CONSTRUCTION BY STEPWISE REFINEMENT

In this section, we constructively derive a protocol system that guarantees correct delivery of data, provided that P_1 satisfies two local time constraints. These local time contraints correspond to R_2 and R_3 obtained in Section 2. Thus, the best case and worst case values of $\{a_n\}$ obtained in Section 2 hold for this protocol system, provided it tightly enforces the given time constraints. In Section 5, we specify three different methods by which P_1 can enforce these local time constraints.

The protocol system is constructed in successive steps. At any point in the construction, we have the following:

(a) An *image protocol system*, which corresponds to a part, or more precisely a projection (see below and [9]), of the final protocol system. An image protocol system is a fully specified protocol system in its own right, complete with messages, state vector **v**, Initial(**v**), and events $e_1(\mathbf{v}; \mathbf{v}")$, ..., $e_n(\mathbf{v}; \mathbf{v}")$.

(b) A set of *desired* safety properties A_0, A_1, ..., A_n, which is *partially* satisfied by the current image protocol. More precisely, let **A** denote the conjunction of all the A_i's. Then, we have the following (see safety inference rule): Initial $\Rightarrow$ **A**; and for each A_i, (**A and** e) $\Rightarrow$ A_i" holds for some subset (which may be the empty or total set) of the events.

We start the construction with a very simple image protocol that merely formalizes the English description of the protocol given in paragraphs 2 and 3 in Section 1. The initial set of desired safety properties consists only of statements defining correct data transfer and correct interpretation of received sequence numbers. None of these properties are satisfied by the initial image protocol.

Each step in the protocol construction is one of two types. The first type of construction step modifies the events and set of desired properties without introducing any new state variables. Here, we choose an event e and a desired safety property A_i that is not preserved by the event; i.e., (**A and** e) $\Rightarrow$ A_i" does not hold. Then we determine the precondition p(**v**) [4], which if it held before the occurrence of e, is sufficient to enforce A_i after the event occurrence; i.e., (**A and** p **and** e) $\Rightarrow$ A_i" holds. If e is an event of entity P_i and p is a condition that can be enforced by P_i, then the event e is replaced by the new event $e_{new} \equiv$ (p **and** e). In addition to preserving A_i, this new event may preserve other desired properties in **A**. If (p **and** e) is not implementable as an event, then p is appended to the set of desired safety properties.

The second type of construction step introduces new state variables and modifies existing events to update these new state variables. The addition of new state variables is often needed to prevent certain event sequences from occurring. For example, given two events e_1 and e_2 of P_1, if we wish e_2 to occur within some specified time of e_1's occurrence, then we need a timer that measures the time elapsed following e_1's occurrence.

In either type of step, whenever an old event e_{old} is modified to a new event e_{new}, we insist that $e_{new} \Rightarrow e_{old}$ must hold; i.e., the effect of e_{new} on the state variables that existed prior to the modification is a special case of the effect of e_{old} on those variables. e_{new} is referred to as a *refinement* of e_{old}. Clearly, for any property A_i and any event e, including e_{old}, if (**A and** e) $\Rightarrow$ A_i" held before, it continues to hold for the new image protocol. The construction terminates when all the desired safety properties are verified; i.e., (**A and** e) $\Rightarrow$ **A**" for all events.

4.1 Initial image protocol

Let DataSet be the set of data blocks that can be sent in this protocol. P_1 sends messages of type (D,data,ns) where D identifies the type of message, data is a data block from DataSet, and ns is a sequence number. P_2 sends messages of type (ACK,nr) where nr is a sequence number. ns and nr are restricted to the values 0,1,...,N-1, where $N \geq 2$.

For purposes of specifying correct interpretation of received sequence numbers, we include in each message an auxiliary field that equals the unbounded sequence number which the cyclic sequence number in the message is attempting to identify. Each D message now has the form (D,data,ns,uns) where uns records the data block number of the data block in the message. Each ACK message now has the form (ACK,nr,unr) where unr records the data block number that was next expected at P_2 when the ACK message was sent. The uns and unr fields are strictly for purposes of discussion, and are not available to the protocol entity implementions.

During the course of the protocol construction, if we generate a desired safety property that is in fact invariant for the current image protocol (i.e. preserved by all the events), then we shall label the assertion as B_i, i=0,1,..., rather than as A_i.

Give integers i and j, the notation [i..j] denotes the sequence (i, i+1, ..., j) if $i \leq j$, and the null sequence if $i > j$. The Pascal-like notation S : array[0..s-1] of T defines S to be the variable sequence (S[0], S[1], ..., S[s-1]), where each S[i] is a value from T, and s is the length of the sequence. The notation S[i..j] denotes the subsequence (S[i], S[i+1], ..., S[j]) if $i \leq j$, and the null sequence if $i > j$. The shorthand notation S[i..j]=t, where t is a value in T, denotes (for every n in [i..j])[S[n]=t].

We use $\oplus$ and $\ominus$ to denote modulo-N addition and subtraction respectively.

Finally, for brevity in stating an event predicate, we use the following guarded command [4] notation $e_1(v;v'') \rightarrow e_2(v;v'')$ to mean that the action in e_2 is done only if e_1 is enabled. Formally, $e_1(v;v'') \rightarrow e_2(v;v'')$ expands to the following:

$e_1(v;v'') \Rightarrow e_2(v;v'')$ **and** (not $e_1(v;v'') \Rightarrow$(for every v" in the body of e_2)[v"=v])

Entity P_1

At P_1, define the following auxiliary variables:

Source : array[0..s-1] of DataSet; {Source is an auxiliary history variable that records the sequence of data blocks accepted by P_1 from its local source. s indicates the length of Source. Initially, s=0 and Source is the null sequence.}

a : $0..\infty$; {Auxiliary variable indicating the number of data blocks that have been acknowledged; i.e., Source[0..a-1] are acknowledged and Source[a..s-1] are unacknowledged. a is initialized to 0 and is always less than or equal to s.}

and the following implemented variables:

vs : 0..N-1; {Initialized to 0 and always equals s mod N}

va : 0..N-1; {Initialized to 0 and always equals a mod N}

Unacknowledged data blocks are saved in buffers local to P_1. Therefore, even though Source, a, s are auxiliary variables, the implementation has access to the value s-a and the data blocks Source[a+i] for any i in [0..s-a-1].

We now specify the events of P_1. Acceptance of a new data block from the local source is formally modeled by the following internal event:

AcceptData($\mathbf{v}_1$;$\mathbf{v}_1''$) ≡ Source[s]" in DataSet {accept data}
and s"=s+1 **and** vs"=vs⊕1 {update state}

Notice that this event does not incorporate the send window size SW. SW will be introduced when it is needed in the construction (Section 4.8).

Transmission of a D message containing an unacknowledged data block is modeled by the following send event:

Send_D ($\mathbf{v}_1$, $\mathbf{z}_1$; $\mathbf{v}_1''$, $\mathbf{z}_1''$)
≡ (for some i in [0..s-a-1])[Send_1 ((D,Source[a+i],va⊕i,a+i),$\mathbf{z}_1$; $\mathbf{z}_1''$)]

Note that this event allows P_1 to send any unacknowledged data block at any time.

Reception of an ACK message is modeled by the following receive event (note that this event does not use the unr field in the ACK message):

Rec_ACK ($\mathbf{v}_1$, $\mathbf{z}_2$; $\mathbf{v}_1''$, $\mathbf{z}_2''$)
≡ (for some nr in [0..N-1])(for some integer unr)
[Rec_2 ($\mathbf{z}_2$; (ACK,nr,unr), $\mathbf{z}_2''$)
and (for some i in [1..s-a])[va⊕i=nr → (va"=va⊕i **and** a"=a+i)]]

Note that we do not distinguish between those unacknowledged messages which have not yet been sent and those that have already been sent. This causes no problems because we allow P_1 to send any unacknowldeged message at any time.

Entity P_2

At P_2, define the following auxiliary variables:

Sink : array[0..∞] of DataSet ∪ {empty}; {Sink is an auxiliary history variable that records the data blocks which were received by P_2 and not discarded. Initially, Sink[0..∞]=empty}

r : 0..∞; {Sink[0..r-1] is the sequence of data blocks that have been passed on to the destination, and r is the number of such data blocks. Sink[r..r+RW-1] correspond to buffers within P_2 used for storing data blocks received out-of-sequence. Initially, r=0}

and the following implemented variable:

vr : 0..N-1; {Initialized to 0 and always equals r mod N}

We now specify the events of P_2. Reception of a D message is modeled by the following receive event (note that this event does not use the uns field in the D message):

Rec_D ($\mathbf{v}_2$, $\mathbf{z}_1$; $\mathbf{v}_2''$, $\mathbf{z}_1''$)
≡ (for some data in DataSet)(for some ns in [0..N-1])(for some integer uns)
[Rec_1($\mathbf{z}_1$; (D,data,ns,uns), $\mathbf{z}_1''$) **and**
(for some i in [0..RW-1])
[(vr⊕i=ns {if ns falls in the receive window}
and Sink[r+i]=empty) {and the corresponding buffer is empty}
→ (Sink[r+i]" = data {then sink the received data}
and (i=0 → SinkData(r,vr; r",vr")))]] {if received data is next expected then pass data blocks to destination}

where

SinkData(r,vr; r",vr")
≡ (for some k in [1..RW])
[Sink[r+k]=empty {Sink[r+k] is the first empty buffer}
and (for all j in [1..k-1])[Sink[r+j]≠empty]
and r"=r+k **and** vr"=vr⊕k] {pass buffered data in Sink[r..r+k-1] to destination}

It is necessary that RW≥1, otherwise every received data block will be discarded.

Transmission of an ACK message is modeled by the following send event:

Send_ACK($\mathbf{v}_2$, $\mathbf{z}_2$; $\mathbf{v}_2''$, $\mathbf{z}_2''$)
≡ Send_2((ACK,vr,r), $\mathbf{z}_2$; $\mathbf{z}_2''$)

In this initial image protocol, P_2 can send an ACK message at any time.

Other events

For i = 1 and 2, the channel events of C_i are specified by a predicate ChannelError($\mathbf{z}_i$; $\mathbf{z}_i''$) that allows all possible losses, duplications and reorderings of <message,age> pairs in the channel. The timer axiom for C_i is $\text{TimerAxiom}_i(\mathbf{z}_i)$ ≡ (for all <m,t> in $\mathbf{z}_i$)[$0 \leq t \leq \text{MaxDelay}_i$].

The *ideal time event* is specified by
$\eta'' = \eta+1$ **and** $\mathbf{z}_1''$=next($\mathbf{z}_1$) **and** $\mathbf{z}_2''$=next($\mathbf{z}_2$)
and $\text{TimerAxiom}_1(\mathbf{z}_1'')$ **and** $\text{TimeAxiom}_2(\mathbf{z}_2'')$

Since there are no local timer variables in this initial image protocol, we have no local time events.

4.2 Desired safety properties

We now formally specify the desired safety properties of correct data transfer and correct interpretation of received sequence numbers. These properties are not satisfied by the current image protocol. We also specify some trivial but related safety properties that are satisfied by the current image protocol.

Correct data transfer

Correct data transfer is formally specified by requiring the following to be invariant:

A_0 (for all n in [0..r-1])[Sink[n] = Source[n]]

A_1 $0 \leq a \leq r \leq s$

A_0 states that data is delivered to the destination at P_2 in the same order as it was accepted from the source at P_1. A_1 states that data is acknowledged at P_1 only after it has been delivered to the destination at P_2, which in turn happens only after it was accepted from the source at P_1. (See Figure 2.)

We also expect that the out-of-sequence data blocks buffered at P_2 have been identified correctly; otherwise A_0 will be violated when these blocks are passed to the destination. Formally, we desire the following to be invariant:

A_2 (for all n in [r+1..r+RW-1])[Sink[n]$\neq$empty $\Rightarrow$ Sink[n]=Source[n]]

Because P_2 does not accept data blocks outside its receive window, and because Source[s-1] is the highest numbered data block in Source, $A_{0\text{-}2}$ implies the following (the notation $A_{0\text{-}2}$ denotes (A_0 **and** A_1 **and** A_2)):

A_3 (for all n in [min(s, r+RW)..∞])[Sink[n] = empty]

Correct interpretation of received D messages

The Rec_D event correctly interprets the sequence number ns in a received (D,data,ns,uns) message if it does the following: if uns is within the receive window [r..r+RW-1] then r+i (in the Rec_D event) should equal uns; if uns is outside the receive window then the Rec_D event should ignore the data block. Because of channel errors, any D message in C_1 can be received by P_2. Thus, a necessary and sufficient condition for correct interpretation of D messages is that the following should be invariant:

A_4 (D,data,ns,uns) in $\mathbf{z}_1$ $\Rightarrow$
((uns in [r..r+RW-1] $\Rightarrow$ (for exactly one i in [0..RW-1])[vr$\oplus$i=ns **and** uns=r+i])
and (uns not in [r..r+RW-1] $\Rightarrow$ (for all i in [0..RW-1])[vr$\oplus$i$\neq$ns]))

Correct interpretation of received ACK messages

The Rec_ACK event correctly interprets the sequence number nr in a received (ACK,nr,unr) message if it does the following: if unr is in [a+1..s] (i.e., acknowledges data that is currently unacknowledged) then Rec_ACK should update a to equal unr; if unr is not in

[a+1..s] then the ACK message should be ignored. Because any ACK message in C_2 can be received by P_1, a necessary and sufficient condition for correct interpretation of ACK messages is that the following should be invariant:

A_5 (ACK,nr,unr) in $\mathbf{z}_2$ ⇒
((unr in [a+1..s] ⇒ (for exactly one i in [1..s-a])[va⊕i=nr **and** unr=a+i])
and (unr not in [a+1..s] ⇒ (for all i in [1..s-a])[va⊕i≠nr]))

Some desired safety properties that trivially hold

Because P_2 does not buffer insequence data blocks, it is obvious that the following is invariant (the proof is trivial and is given below):

B_0 Sink[r]=empty

Proof of invariance of B_0.

We shall establish the invariance of B_0 by proving that the following hold (recall the safety inference rule): Initial⇒B_0, and (B_0 **and** e)⇒B_0" for every event e. Each of these statements has the form *C*⇒*D*. Its proof will be presented as a sequence of steps, each consisting of a statement *L* at the left and a list of statements R_1,R_2, ..., R_n at the right. *L* derives (in the predicate calculus augmented with axioms for the data types) from (R_1 **and** ... **and** R_n). Each R_i is implied either by an *L* derived in an earlier step, or by *C*. The conjunction of the *L*'s imply *D*. Finally, we say that event e *does not affect D* if e implies that v"=v for every variable v in *D*. Clearly, if e does not affect *D* then (e **and** *D*)⇒*D*" holds.

Initial Conditions:
(a) B_0 — (r=0, Sink[0..RW-1]=empty)
Consider Rec_D:
(a) ns≠vr ⇒ (r"=r **and** Sink[r]=Sink[r]") ⇒ B_0" — (Rec_D, B_0)
(b) ns=vr ⇒ (r">r **and** Sink[r"]"=empty) ⇒ B_0" — (Rec_D)
(c) B_0" — (a, b)
Any event other than Rec_D does not affect B_0.
End of proof of B_0

The following invariant states that the modulo-N counters correctly track the corresponding unbounded counts:

B_1 vs=s mod N **and** va=a mod N **and** vr=r mod N

Proof of invariance of B_1.
Each conjunct of B_1 individually satisfies the safety inference rule. We will give details only for the conjunct vs=s mod N.

Initial Conditions:
(a) vs=s mod N — (vs=s=0)
AcceptData:
(a) vs"=s" mod N — (s"=s+1, vs"=vs⊕1, vs=s mod N)

Any event other than AcceptData does not affect vs=s mod N.
End of proof of B_1

4.3 Refinement of A_4 and Rec_D

We now consider additional properties of D messages in C_1. First, note that the following invariant is obviously satisfied by the current image protocol (formal proof of invariance is in Appendix A):

B_2 (D,data,ns,uns) in $\mathbf{z}_1$ ⇒ (uns in [0..s-1] **and** data=Source[uns] **and** ns=uns mod N)

Second, note that in order for A_4 to hold, it is necessary that

B_3 $1 \leq \text{RW} \leq \text{N}$

Otherwise, for a (D,data,ns,uns) message with uns=r, we will have two values, 0 and N, in [0..RW-1] such that vr⊕0=vr⊕N=ns. No verification is needed for B_3 because RW and N are constants.

Given 1≤RW≤N and i in [0..RW-1], we have (vr⊕i=ns ⇔ i=ns⊖vr). Thus, the Rec_D event can be refined to the following:

Rec_D($\mathbf{v}_2$,$\mathbf{z}_1$;$\mathbf{v}_2''$,$\mathbf{z}_1''$)
  ≡ (for some data in DataSet)(for some ns in [0..N-1])(for some integer uns)
    [Rec_1 ($\mathbf{z}_1$; (D,data,ns,uns), $\mathbf{z}_1''$) **and**
    ((0≤ns⊖vr≤RW-1 **and** Sink[r+ns⊖vr]=empty)
      → (Sink[r+ns⊖vr]"=data **and** (ns=vr → SinkData(r,vr; r",vr"))))]

Third, because P_1 can send any outstanding data block at any time, we expect that C_1 can contain (D,data,ns,uns) messages where data is any block from Source[a..s-1]. In addition to these D messages, we expect that C_1 can contain older D messages with earlier data blocks that are currently not outstanding. Let Source[l_1] denote the earliest such data block, and let us assume that C_1 may contain a (D,data,ns,uns) message for any uns in [l_1..s-1]. (See Figure 3.) The following lemma implies that correct interpretation, i.e. A_4, holds if s-1 does not exceed r+N-1 and if l_1 does not lag behind r-N+RW.

Lemma 1. Given 1≤RW≤N and ns=uns mod N, the following hold:

(a) uns in [r+RW..r+N-1] ⇒ RW≤ns⊖vr≤N-1

(b) uns in [r..r+RW-1] ⇒ (0≤ns⊖vr≤RW-1 **and** uns=r+ns⊖vr)

(c) uns in [r-N+RW..r-1] ⇒ RW≤ns⊖vr≤N-1

(d) uns=r-N+RW-1 ⇒ (0≤ns⊖vr≤RW-1 **and** uns≠r+ns⊖vr=r+RW-1)

(e) uns=r+N $\Rightarrow$ (0$\leq$ns$\ominus$vr$\leq$RW-1 **and** uns$\neq$r+ns$\ominus$vr=r)

Proof of Lemma 1.

(a) uns in [r+RW..r+N-1] $\Rightarrow$ uns-r in [RW..N-1] $\Rightarrow$ ns$\ominus$vr = uns-r $\Rightarrow$ ns$\ominus$vr in [RW..N-1]

(b) uns in [r..r+RW-1] $\Rightarrow$ uns-r in [0..RW-1] $\Rightarrow$ ns$\ominus$vr = uns-r $\Rightarrow$ ns$\ominus$vr in [0..RW-1] **and** uns = r+ns$\ominus$vr.

(c) uns in [r-N+RW..r-1] $\Rightarrow$ uns-r in [-N+RW..-1]. Since 0 < RW $\leq$ N, [-N+RW..-1] $\subseteq$ [-N+1..-1] and ns$\ominus$vr = uns-r+N. Hence, ns$\ominus$vr is in [RW..N-1].

(d) uns=r-N+RW-1 $\Rightarrow$ uns-r=-N+RW-1 $\Rightarrow$ ns$\ominus$vr = RW-1. Thus, ns$\ominus$vr < RW and r + ns$\ominus$vr = r+RW-1 $\neq$ uns.

(e) uns=r+N $\Rightarrow$ uns-r=N $\Rightarrow$ ns$\ominus$vr = 0 < RW and r+ns$\ominus$vr = r$\neq$uns.

End of proof of Lemma 1

Thus, correct interpretation of received D messages is guaranteed if the following is invariant:

A_6 (D,data,ns,uns) in $\mathbf{z}_1$ $\Rightarrow$ uns in [r-N+RW..r+N-1]

4.4 Refinement of A_5 and Rec_ACK

We now consider additional properties of ACK messages in C_2. First, note that the following invariant is obviously satisfied by the current image protocol (formal proof of invariance in Appendix A):

B_4 (ACK,nr,unr) in $\mathbf{z}_2$ $\Rightarrow$ (unr in [0..r] **and** nr=unr mod N)

Second, note that in order for A_5 to hold, it is necessary that following is invariant:

A_7 s-a $\leq$ N-1

Otherwise, for an (ACK,nr,unr) message with unr=a, we will have N in [1..s-a] such that va$\oplus$N=nr.

Given s-a$\leq$N-1 and i in [1..s-a], we have s-a=vs$\ominus$va, and (va$\oplus$i=nr $\Leftrightarrow$ i=nr$\ominus$va). Thus, the Rec_ACK event can be refined to the following:

Rec_ACK($\mathbf{v}_1$, $\mathbf{z}_2$; $\mathbf{v}_1''$, $\mathbf{z}_2''$)
  $\equiv$ (for some nr in [0..N-1])(for some integer unr)
    [Rec_2($\mathbf{z}_2$; (ACK,nr,unr), $\mathbf{z}_2''$)
    and (1$\leq$nr$\ominus$va$\leq$vs$\ominus$va $\rightarrow$ (va"=nr **and** a"=a+nr$\ominus$va))]

Third, we expect C_2 to contain (ACK,nr,unr) messages that carry new information to P_1, i.e., with unr in [a+1..r]. We also expect C_2 to contain older ACK messages with unr$\leq$a. Let l_2 denote the lowest value of unr in C_2, and let us assume that C_2 may contain an (ACK,nr,unr) message for any unr in [l_2..r]. (See Figure 3.) The following lemma implies that correct interpretation, i.e. A_5, holds if r does not exceed s and if l_2 does not lag behind s-N+1.

Lemma 2. Given $0 \leq a \leq s \leq a+N-1$ and nr=unr mod N, the following hold:

(a) unr in [a+1..s] $\Rightarrow$ (nr$\ominus$va in [1..vs$\ominus$va] **and** unr=a+nr$\ominus$va)

(b) unr in [s-N+1..a] $\Rightarrow$ nr$\ominus$va not in [1..vs$\ominus$va]

(c) (unr=s-N **and** vs$\neq$va) $\Rightarrow$ (nr$\ominus$va in [1..vs$\ominus$va] **and** unr$\neq$a+nr$\ominus$va =s)

Proof of Lemma 2.

(a) unr in [a+1..s] $\Rightarrow$ (unr-a) in [1..s-a] $\Rightarrow$ unr-a = nr$\ominus$va

(b) unr=a $\Rightarrow$ unr-a = nr$\ominus$va=0 which is not in [1..vs$\ominus$va]. unr in [s-N+1..a-1] $\Rightarrow$ (unr-a) in [s-a-N+1..-1]. Since s-a$\leq$N-1, we have [s-a-N+1..-1] $\subseteq$ [-N+1..-1]. Hence, nr$\ominus$va = unr-a+N. Hence, nr$\ominus$va in [s-a+1..N-1] = [vs$\ominus$va + 1..N-1].

(c) unr=s-N $\Rightarrow$ unr-a=s-a-N $\Rightarrow$ nr$\ominus$va = vs$\ominus$va = s-a. Thus a+nr$\ominus$va = s $\neq$ unr, but nr$\ominus$va in [1..vs$\ominus$va] since vs$\ominus$va $\neq$ 0.

End of proof of Lemma 2

Thus, correct interpretation of received ACK messages is guaranteed if ((ACK,nr,unr) in $\mathbf{z}_2$ $\Rightarrow$ unr in [s-N+1..s]) is invariant. Combining this with $A_1 \Rightarrow r \leq s$, we have that the following should be invariant:

A_8 (ACK,nr,unr) in $\mathbf{z}_2$ $\Rightarrow$ unr in [s-N+1..r]

4.5 Refinement of A_{6-8} and the AcceptData event

Observe that A_{0-3} can be violated only if received messages are interpreted incorrectly. In Section 4.3, we have shown that conditions B_{2-3} and A_6 imply A_4, i.e., ensure correct interpretation of D messages. In Section 4.4, we have shown that conditions A_{7-8} imply A_5, i.e., ensure correct interpretation of ACK messages. Therefore, the protocol construction problem reduces to finding ways to ensure that A_{6-8} are invariant (B_{2-3} have already been proved to be invariant). For each event e(v;v") of the current image protocol, we next determine what conditions must hold before the event occurrence, in order that A_{6-8} holds after the event occurrence. In each case, we assume that A_{0-8} holds before the event occurrence.

First, consider the Rec_D event. The occurrence of this event can increase the value of state variable r; i.e., if r" denotes the value of the state variable after the event occurrence, and

r denotes the value before the event occurrence, then $r" \geq r$. This does not affect A_7. A_8 holds after the event occurence if it held prior to the event occurrence. In order that A_6 hold after the event occurrence, it is necessary that ((D,data,ns,uns) in $\mathbf{z}_1 \Rightarrow uns \geq r"-N+RW$) hold before the event occurrence. Note that r" can be any value in [r..s], and any D message in $\mathbf{z}_1$ can be received at any time. Therefore, it is necessary that ((D,data,ns,uns) in $\mathbf{z}_1 \Rightarrow uns \geq s-N+RW$) be invariant. Also, observe that $A_{1,7} \Rightarrow r+N-1>s-1$ (the notation $A_{1,7}$ denotes A_1 **and** A_7). Combining the above with A_6 and B_2, we can state that the following should be invariant.

A_9 (D,data,ns,uns) in $\mathbf{z}_1 \Rightarrow$ uns in [s-N+RW..s-1]

Because $A_{9,7,1}$ implies A_6, we will no longer consider A_6.

Next, we consider the Send_D event. This event introduces into C_1 a (D,data,ns,uns) message, where uns can be any value in [a..s-1]. While this does not violate A_7 or A_8, it can violate A_9. In order for A_9 to hold after the send, it is necessary that $a \geq s-N+RW$ must hold before the send. Because the send can occur whenever s>a, and $a \geq s-N+RW$ holds whenever s=a, we require that the following be invariant.

A_{10} $a \geq s-N+RW$

Because $RW \geq 1$ and A_{10} imply A_7, we will no longer consider A_7.

Send_ACK and Rec_ACK do not violate $A_{8\text{-}10}$.

Finally, consider the AcceptData event. In order that $A_{8\text{-}10}$ hold after the event occurrence, it is necessary that the following hold prior to the event occurrence:

E_0 $s-a \leq N-RW-1$
E_1 (D,data,ns,uns) in $\mathbf{z}_1 \Rightarrow uns \geq s-N+RW+1$
E_2 (ACK,nr,unr) in $\mathbf{z}_2 \Rightarrow unr \geq s-N+2$

E_0, E_1 and E_2 are obtained from A_{10}, A_9 and A_8 respectively, by replacing s with s+1.

Observe that E_0 can be incorporated into the enabling condition of the AcceptData event, because s-a is available to the implementation of P_1. In fact, because $A_{10} \Rightarrow s-a=vs \ominus va$, we can refine the AcceptData event to the following:

AcceptData($\mathbf{v}_1$;$\mathbf{v}_1$")
$\equiv vs \ominus va \leq N-RW-1 \rightarrow$ (Source[s]" in DataSet **and** s"=s+1 **and** vs"=vs$\oplus$1)

Unlike condition E_0, the conditions E_1 and E_2 cannot be included in the enabling condition of AcceptData. This is because the implementation at P_1 does not have access to the messages in the channels. In Sections 4.6 and 4.7, we show how E_1 and E_2 can be implemented by exploiting the bounded message lifetime property of the channels.

The discussion above is formalized in the following lemma (formal proof in Appendix A):

Lemma 3.

(a) $A_{0\text{-}3,8\text{-}10}$ hold initially

(b) (for every event e other than AcceptData)[($A_{0\text{-}3,8\text{-}10}$ **and** $B_{0\text{-}4}$ **and** e) $\Rightarrow$ $A''_{0\text{-}3,8\text{-}10}$]

(c) ($A_{0\text{-}3,8\text{-}10}$ **and** $B_{0\text{-}4}$ **and** AcceptData) $\Rightarrow$ ($A''_{0\text{-}3}$ **and** A_{10}'')

(d) (E_2 **and** A_8 **and** AcceptData) $\Rightarrow$ A_8''

(e) (E_1 **and** A_9 **and** AcceptData) $\Rightarrow$ A_9''

Observe that if RW=N, then $A_{1,10}$ implies that AcceptData is never enabled. To avoid this safe but dead protocol, we must restrict RW to satisfy the following:

B_5 $\quad 1 \leq \text{RW} \leq \text{N-1}$

In fact, if RW=N and C_1 has D messages containing Source[n] with n=r, then the following incorrect behavior can occur: P_2 receives Source[n], places it in Sink[n], and advances r to n+1; P_2 then receives a duplicate Source[n] and places it in Sink[n+N]!

4.6 An implementable local time constraint that enforces E_1

In this section, we derive a time constraint that P_1 can implement which guarantees that E_1 holds prior to accepting a new data block. The derived time constraint will be expressed in terms of ideal timers. In Section 5, we present three different local timer implementations of this time constraint.

For any uns in [0..s-N+RW], there are exactly two ways in which P_1 can be sure that C_1 does not contain a (D,data,ns,uns) message: either a (D,data,ns,uns) message has never been sent, or more than MaxDelay_1 time units has elapsed since a (D,data,ns,uns) message was last sent. With this motivation, we define the following array of auxiliary ideal timers at P_1:

DTimeG : array[0..∞] of (Off,0,1,...); {DTimeG[n] indicates the ideal time elapsed since Source[n] was last sent. DTimeG[n]=Off if Source[n] has never been sent. Initially, DTimeG[0..∞]=Off}

DTimeG is reset in the Send_D event, which is now refined to the following:

Send_D ($\mathbf{v}_1$;$\mathbf{z}_1$;$\mathbf{v}_1''$,$\mathbf{z}_1''$)
  ≡ (for some i in [0..s-a-1])
    [Send_1((D,Source[a+i],va$\oplus$i,a+i), $\mathbf{z}_1$;$\mathbf{z}_1''$) **and** DTimeG[a+i]"=0]

DTimeG is aged by the ideal time event, which now has DTimeG"=next(DTimeG) as a conjunct.

The following invariant property is obviously satisfied by the current image protocol (formal proof in Appendix A):

B_6 $<$(D,data,ns,uns), age$>$ in $\mathbf{z}_1 \Rightarrow$ age$\geq$DTimeG[n]$\geq$0

It is obvious from B_6 and the timer axiom of C_1 that E_1 holds if the following local time constraint holds:

E_3 (for all n in [0..s-N+RW])[DTimeG[n]>$MaxDelay_1$ **or** DTimeG[n]=Off]

(In fact, because of $A_{0\text{-}3,10}$, we can prove that DTimeG[0..s-N+RW]$\geq$$MaxDelay_1$ should hold; however, we shall see that E_3 is sufficient for our purposes.)

To directly enforce E_3, P_1 would need an unbounded number of local timers, one to track each ideal timer in DTimeG[0..s-1]. We now consider ways to enforce E_3 by using a bounded number of local timers. Assume that E_3 has held prior to every past occurrence of AcceptData. In particular, if s_0 denotes the current value of state variable s, then (for all n in [0..s_0-1-N+RW])[DTimeG[n]>$MaxDelay_1$ **or** DTimeG[n]=Off] held just prior to the last occurrence of AcceptData. We also know that no data block in Source[0..s_0-N+RW-1] was sent after that instant, because a$\geq$$s_0$-1+1-N+RW held at that instant. Thus, we expect that the following is invariant:

A_{11} (for all n in [0..s-N+RW-1])[DTimeG[n]>$MaxDelay_1$ **or** DTimeG[n]=Off]

Thus, to enforce E_3 it is sufficient to enforce the following local time constraint:

E_4 s$\geq$N-RW $\Rightarrow$ (DTimeG[s-N+RW]>$MaxDelay_1$ **or** DTimeG[s-N+RW]=Off)

The above discussion is formalized in the following lemma (formal proof in Appendix A):

Lemma 4.

(a) A_{11} holds initially

(b) (for every event e other than AcceptData)[($A_{1,10,11}$ **and** e) $\Rightarrow$ A''_{11}]

(c) ($A_{9,11}$ **and** B_6 **and** AcceptData **and** E_4) $\Rightarrow$ (A''_{11} **and** E_1)

To enforce E_4, it is sufficient if P_1 tracks the ideal timers in DTimeG[s-N+RW..s-1]. This can be done with a bounded number of local timers, each of bounded counter capacity. For instance, P_1 can use a circular array of N-RW local timers; local timer n mod N-RW can track DTimeG[n], for n in [max(0, s-N+RW)..s-1]. Each local timer can be reset to Off once it indicates that the corresponding ideal timer has exceeded $MaxDelay_1$. (See Section 5 for formal specifications of different implementations.)

4.7 An implementable local time constraint that enforces E_2

We now consider how P_1 can ensure that condition E_2 holds before accepting a new data block. For any unr in [0..s-N+1], there are exactly two ways in which we can be sure that C_2 does not contain an (ACK,nr,unr) message: either (ACK,nr,unr) has never been sent, or more than $MaxDelay_2$ time units has elapsed since (ACK,nr,unr) was last sent. Unlike the previous case which involved D messages, P_1 does *not* have access to the time elapsed since (ACK,nr,unr) was last sent. This is because ACK messages are sent by P_2 and not by P_1. However, P_1 can obtain a lower bound on this elapsed time because of the following considerations: P_2 does not send an (ACK,nr,unr) message with unr=n once r exceeds n; also, from A_1 we see that a exceeds n only after r exceeds n. Thus, the time elapsed since a exceeded n is a lower bound on the ages of all (ACK,nr,n) in C_2. Furthermore, this elapsed time can be measured by P_1. E_2 holds whenever more than $MaxDelay_2$ time has elapsed since a exceeded s-N+1.

With this motivation, define the following array of auxiliary ideal timers at P_2:

RTimeG : array[0..∞] of (Off,0,1,...); {RTimeG[n] indicates the ideal time elapsed since **r** first exceeded n. Initially, RTimeG[0..∞]=Off}

RTimeG is reset in the SinkData predicate of the Rec_D event. The SinkData predicate is now refined to the following:

SinkData($\mathbf{v}_2$;$\mathbf{v}_2''$)
≡ (for some k in [1..RW])[Sink[r+k]=empty
and (for all j in [1..k-1])[Sink[r+j]≠empty **and** RTimeG[r+j]"=0]
and r"=r+k **and** vr"=vr⊕k]

RTimeG is aged by the ideal time event which now has RTimeG"=next(RTimeG) as a conjunct.

The following invariant properties are obviously satisfied by the current image protocol (formal proof in Appendix A):

B_7 RTimeG[0] ≥ RTimeG[1] ≥ ... ≥ RTimeG[r-1] ≥ 0
B_8 <(ACK,nr,unr), age> in $\mathbf{z}_1$ ⇒
((unr<r ⇒ age≥RTimeG[unr]) **and** (unr≥1 ⇒ age≤RTimeG[unr-1]))

Next, define the following array of auxiliary ideal timers at P_1:

ATimeG : array[0..∞] of (Off,0,1,...); {ATimeG[n] indicates the ideal time elapsed since a first exceeded n. Initially, ATimeG[0..∞]=Off}

ATimeG is reset in the Rec_ACK event which is now as follows:

Rec_ACK($\mathbf{v}_1$,$\mathbf{z}_2$;$\mathbf{v}_1''$,$\mathbf{z}_2''$)
≡ (for some nr in [0..N-1])(for some integer unr)

[$Rec_2(\mathbf{z}_2;(ACK,nr,unr),\mathbf{z}_2'')$
and ($1 \leq nr \ominus va \leq vs \ominus va \rightarrow (va''=nr$ **and** $a'' = a + nr \ominus va$
and (for all i in [0..nr⊖va-1])[ATimeG[a+i]"=0]))]

ATimeG is aged by the ideal time event which now has ATimeG"=next(ATimeG) as a conjunct.

The following invariant property is obviously satisfied by the current image protocol (formal proof in Appendix A):

B_9 ATimeG[0] $\geq$ ATimeG[1] $\geq$ ... $\geq$ ATimeG[a-1] $\geq$ 0

Because a exceeds n only after r exceeds n, we expect the following to be invariant:

A_{12} (for all n in [0..a-1])[ATimeG[n] $\leq$ RTimeG[n]]

From $B_{7\text{-}9}$, the timer axiom for C_2, and A_{12}, we see that E_2 is implied by the following local time constraint:

E_5 $s \geq N\text{-}1 \Rightarrow$ ATimeG[s-N+1]>$MaxDelay_2$

Because each ATimeG[n] is reset only once, if E_5 has held for all previous AcceptData event occurrences, then it is obvious that the following is invariant:

A_{13} (for all n in [0..s-N])[ATimeG[n] > $MaxDelay_2$]

The above discussion is formalized in the following lemma (formal proof in Appendix A):

Lemma 5.

(a) $A_{12\text{-}13}$ hold initially

(b) (for every event e other than AcceptData)[($A_{1,10,12\text{-}13}$ **and** B_7 **and** e) $\Rightarrow A''_{12\text{-}13}$]

(c) ($A_{12\text{-}13}$ **and** $B_{7\text{-}9}$ **and** AcceptData **and** E_5) $\Rightarrow$ ($A''_{12\text{-}13}$ **and** E_2)

A_{13} implies that P_1 can enforce E_5 by tracking the ideal timers in ATimeG[s-N+1..a-1]. For example, P_2 can do this with a circular array of N-1 local timers where local timer n mod N-1 tracks ATimeG[n] for n in [max(0, s-N+1)..a-1]. Note that each local timer can be reset to Off once it indicates that the corresponding ideal timer has exceeded $MaxDelay_2$. (See Section 5 for formal specifications of different implementations.)

4.8 The final protocol assuming the enforcement of E_4 and E_5

In Sections 4.6 and 4.7, we have shown that local time constraints E_4 and E_5 enforce E_1 and E_2 respectively (Lemmas 4 and 5). Combining this with Lemma 3 from Section 4.5, we see

that any image protocol that enforces $E_{4,5}$ before accepting new data satisfies the desired safety properties. More formally, we have the following theorem (recall that $A_{0-4,8-10}$ implies A_{5-7}):

Theorem 3.

(a) (A_{0-13} **and** B_{0-9}) hold initially

(b) (for every event e other than AcceptData)[(A_{0-13} **and** B_{0-9} **and** e) $\Rightarrow$ (A''_{0-13} **and** B_{0-9}'')]

(c) (A_{0-13} **and** B_{0-9} **and** AcceptData **and** $E_{4,5}$) $\Rightarrow$ (A''_{0-13} **and** B_{0-9}'')

Two minor modifications that do not affect correctness

We now restrict the transmission of ACK messages by P_2, and restrict the send window size at P_1. These minor modifications are to make the protocol system more realistic, unlike the earlier event modifications which were needed to preserve the invariance of the desired safety properties A_{0-4}.

In the current image protocol, the Send_ACK event is always enabled. We modify P_2 so that it sends an ACK message only as a response to receiving a D message. Define the following variable at P_2:

SendACK : boolean; {True if and only if a D message has been received after the last send of an ACK message. Initially, SendACK=False}

SendACK is set to True in the Rec_D event. The Send_ACK event has SendACK=True as its enabling condition, and its occurrence sets SendACK to False.

The current image protocol has send window size SW equal to N-RW. We now allow SW to satisfy the following:

B_{10} $1 \leq SW \leq N-RW$

Thus, in the AcceptData event, $vs \ominus va \leq N-RW-1$ is now replaced by $vs \ominus va \leq SW-1$. With this refinement to the AcceptData event, it is obvious that the following is invariant:

B_{11} $s-a \leq SW$

Observe that $B_{10,11}$ implies A_{10}. Also, the above modifications are refinements of existing events. Hence, they preserve Theorem 3.

The properties A_{0-13} and B_{0-11} are listed in Table 1. We have grouped them according to the variables that they deal with, rather than list them in the order in which they were introduced. Also, we have omitted those properties which are implied by the properties that are listed.

Lower and upper bounds on performance

We can view the values $\{a_n, l_n, f_n: n \geq 0\}$ from Section 2 as auxiliary variables which are updated in Rec_ACK and Send_D respectively, by being assigned the current value of the ideal time event count η. Because a data block can be sent as soon as it is accepted by P_1, f_n can be treated as the time at which Source[n] is accepted. It is then obvious that E_1, E_4, and E_5 are identical to R_4, R_2 and R_3 respectively. Thus, we have the following:

Theorem 4. Any protocol implementation that tightly enforces the bounds in E_4 and E_5 will achieve the best-case performance in Theorem 1 and the worst-case performance in Theorem 2.

5. COMPLETING THE CONSTRUCTION: THREE WAYS TO ENFORCE E_4 AND E_5

In Section 4, we constructed a protocol system that was complete except for enforcing the conditions E_4 and E_5. In this section, we complete the construction by providing three different methods by which P_1 can enforce E_4 and E_5.

5.1 Protocol implementation which tightly enforces E_4 and E_5

In Section 4.6, we outlined how P_1 can enforce E_4 tightly with a circular array of N-RW local timers. In Section 4.7, we outlined how P_1 can enforce E_5 tightly with a circular array of N-1 local timers. We now provide a formal specification of that implementation, and verify its correctness. For the sake of notational convenience, we consider an implementation with two circular arrays of N local timers, rather than one of N-RW and one of N-1. This allows us to use the existing variables vs and va to index into the arrays, rather than having to define four new counters (indicating s mod N-RW, a mod N-RW, s mod N-1, and a mod N-1) and incorporate modulo N-RW and N-1 arithmetic.

The completed protocol is presented in Tables 2-6. Tables 2 and 4 list the variables of P_1 and P_2 respectively. Tables 3 and 5 list the events of P_1 and P_2 respectively. Table 6 lists the ideal time event and the local time event for P_1. Unlike the entity events, time events are not implemented.

Enforcing E_4

Let $\text{MDelay}_1 = 2 + (1+\epsilon_1)\,\text{MaxDelay}_1$. Define the following array of local timers at P_1:

DTimer : array[0..N-1] of (Off,0,1,...,MDelay_1); {For each n in [max(0, s-N+RW)..s-1], DTimer[n mod N] will track DTimeG[n] upto MDelay_1. Initially, DTimer[0..N-1]=Off}

DTimer[vs$\oplus$i] is reset to 0 whenever Source[a+i] is sent (as shown in the Send_D event in Table 3).

There is now a local time event of P_1 which ages DTimer (as shown in Table 6).

The relationship between DTimer and DTimeG are formalized in the following invariants (formal proof in Appendix B):

B_{12} (for all n in [max(0, s-N+RW)..s-1])
[((DTimer[n mod N], DTimeG[n]) started at 0)
or (DTimer[n mod N]=Off **and** (DTimeG[n]>$MaxDelay_1$ **or** DTimeG[n]=Off))]

B_{13} (for all n in [s..max(s+RW-1, N-1)])[DTimer[n mod N]=Off]

The following lemma holds because (s≥N-RW **and** DTimer[vs⊕RW]=Off **and** B_{12}) ⇒ (DTimeG[s-N+RW]>$MaxDelay_1$ **or** DTimeG[s-N+RW]=Off):

Lemma 6. DTimer[vs⊕RW]=Off ⇒ E_4

Thus, we enforce E_4 by including DTimer[vs⊕RW]=Off in the enabling condition of the AcceptData event (as shown in AcceptData event in Table 3).

Enforcing E_5

Let $MDelay_2 = 2 + (1+\epsilon_1)\ MaxDelay_2$. Define the following array of local timers at $\mathbf{P_1}$:

ATimer : array[0..N-1] of (Off,0,1,...,$MDelay_2$); {For each n in [max(0, s-N+1)..a-1], ATimer[n mod N] will track ATimeG[n] upto $MDelay_2$. Initially, ATimer[0..N-1]=Off}

ATimer[va⊕i] is reset to 0 whenever Source[a+i-1] is acknowledged (as shown in the Rec_ACK event in Table 3).

ATimer is aged by the local time event of P_1, which now has ATimer"=next(ATimer) as a conjunct (as shown in Table 6).

The relationship between ATimer and ATimeG are formalized in the following invariants (formal proof in Appendix B):

B_{14} (for all n in [max(0, s-N+1)..a-1])
[((ATimer[n mod N], ATimeG[n]) started at 0)
or (ATimer[n mod N]=Off **and** ATimeG[n]>$MaxDelay_2$)]

B_{15} (for all n in [a..max(s, N-1)])[ATimer[n mod N] = Off]

The following lemma holds because (s≥N-1 **and** ATimer[vs⊕1]=Off **and** B_{14}) ⇒ ATimeG[s-N+1]>$MaxDelay_2$:

Lemma 7. ATimer[vs⊕1]=Off ⇒ E_5

Thus, E_5 can be enforced by including ATimer[vs⊕1]=Off in the enabling condition of the AcceptData event (as shown in Table 3).

Verification and performance of the implementation

From Theorem 3, Lemmas 6 and 7, and the new AcceptData event, it is obvious that ($A_{0\text{-}13}$ **and** $B_{0\text{-}15}$) is invariant for this protocol implementation.

From $B_{12,14}$ and AcceptData, we see that P_1 can tightly enforce $E_{4,5}$. Thus, by Theorem 4, this protocol implemention satisfies the best-case performance of Theorem 1 and the worst-case performance of Theorem 2. Because the implementation does not rule out any retransmission policy at P_1 or P_2, it can achieve the optimal in the average case too.

5.2 Protocol implementation which loosely enforces E_4 and tightly enforces E_5

Because Source[n] is not sent after it is acknowledged, an alternative way to enforce E_4 is to enforce the following:

E_6 $\quad s \geq N\text{-}RW \Rightarrow \text{ATimeG}[s\text{-}N+RW] > \text{MaxDelay}_1$

E_6 is analogous to E_5, and the implementation can detect it by checking for $\text{ATimer}[vs \oplus RW] > \text{MDelay}_1$. Of course, to enforce E_5, P_1 must still check for $\text{ATimer}[vs \oplus 1] > \text{MDelay}_2$. Thus, we redefine ATimer and AcceptData as follows:

ATimer : array[0..N-1] of max(MDelay_1, MDelay_2)

AcceptData($\mathbf{v}_1$;$\mathbf{v}_1''$)
≡ ($vs \ominus va \leq SW\text{-}1$ **and** ($\text{ATimer}[vs \oplus RW]=\text{Off}$ **or** $\text{ATimer}[vs \oplus RW] > \text{MDelay}_1$)
and ($\text{ATimer}[vs \oplus 1]=\text{Off}$ **or** $\text{ATimer}[vs \oplus 1] > \text{MDelay}_2$))
→ (Source[s]" in DataSet **and** $vs''=vs \oplus 1$ **and** $s''=s+1$)

It is obvious that further simplification results if we consider either the case $\text{MaxDelay}_1 \geq \text{MaxDelay}_2$ or the case $\text{MaxDelay}_1 < \text{MaxDelay}_2$. In the former case, which includes the normal situation of $\text{MaxDelay}_1 = \text{MaxDelay}_2$, the simplification is rather dramatic. We can define ATimer and AcceptData as follows:

ATimer : array[0..N-1] of MDelay_1

AcceptData($\mathbf{v}_1$;$\mathbf{v}_1''$)
≡ ($vs \ominus va \leq SW\text{-}1$ **and** $\text{ATimer}[vs \oplus RW]=\text{Off}$
→ (Source[s]" in DataSet **and** $vs''=vs \oplus 1$ **and** $s''=s+1$)

This protocol implementation is exactly as in Tables 2-6, except for the following changes: the AcceptData event is replaced by one of the above AcceptData events; DTimer is removed from Table 2, the Send_D event in Table 3, and the local time event in Table 6.

Verification and performance of the implementation

The AcceptData event still enforces E_4 and E_5. Hence, from Theorem 3, ($A_{0\text{-}13}$ **and** $B_{0\text{-}11}$) is invariant for this protocol implementation.

The sole difference between this protocol implementation and the one above in Section 5.1 is that this protocol enforces E_6 instead of E_4. This means that instead of $f_n > l_{n\text{-}N+RW} + \text{MaxDelay}_1$ (from R_2), we have $f_n > a_{n\text{-}N+RW} + \text{MaxDelay}_1$. However, $l_n = a_n$ in the best and worst cases. Therefore, by Theorem 4, this protocol implemention satisfies the best-case performance of Theorem 1 and the worst-case performance of Theorem 2. The average performance of this implementation is however worse than the average performance of the implementation in Section 5.1, because $l_n < a_n$ in the average.

5.3 Protocol implementation which loosely enforces E_4 and E_5

Another way to enforce E_4 and E_5 is by restricting the rate at which P_1 accepts new data blocks from its local source. Let the time interval between successive occurences of AcceptData be lower bounded by the constant δ. Note that this time constraint can be implemented with a single local timer at P_1 (below, $\delta_M = 2 + (1+\epsilon_1)\delta$):

STimer : (Off,0,1,...,δ_M); {indicates the local time elapsed, upto δ_M, since the last occurrence of AcceptData. Initially, STimer=Off}

AcceptData($\mathbf{v}_1$;$\mathbf{v}_1''$)
 ≡ (vs⊖va≤SW-1 **and** STimer=Off)
  → (Source[s]" in DataSet **and** vs"=vs⊕1 **and** s"=s+1)

The current protocol implementation is exactly as in Tables 2-6, except for the following changes: STimer is added to Table 2. The AcceptData event is the one specified above. ATimer and DTimer, and all expressions involving them, are deleted from Tables 2 and 3. Note that if δ is sufficiently small, e.g. the hardware clock period, then there is no need for P_1 to explicitly implement STimer.

We now determine the minimum value of δ that will enforce $E_{4,5}$. Define s_n as the time at which Source[n] is accepted by P_1. Because successive occurrences of AcceptData are at least δ apart, we have the following:

R_9 For all n, m: $s_n - s_m \geq (n\text{-}m)\delta$

E_4 corresponds to $s_n - l_{n\text{-}N+RW} \geq \text{MaxDelay}_1$. From R_0, we have $l_{n\text{-}N+RW} \leq a_{n\text{-}N+RW}$. From R_4, we have $a_{n\text{-}N+RW} \leq s_{n\text{-}N+RW+SW}$. Combining these with R_9, we have $s_n - l_{n\text{-}N+RW} \geq (N\text{-}RW\text{-}SW)\delta$. Further, note that equality is achievable in the above bound. Thus, E_4 is enforced if and only if $(N\text{-}RW\text{-}SW)\delta \geq \text{MaxDelay}_1$.

Similarly, E_5 is enforced if and only if $(N\text{-}1\text{-}SW)\delta \geq \text{MaxDelay}_2$. Thus, from Theorem 3 and the new AcceptData event, we have the following:

Theorem 5. ($A_{0\text{-}13}$ and $B_{0\text{-}11}$) is invariant for this protocol implementation if and only if the minimum time δ between accepting new data blocks satisfies the following:

$$\delta \geq \max(\frac{\text{MaxDelay}_1}{\text{N-RW-SW}}, \frac{\text{MaxDelay}_2}{\text{N-1-SW}})$$

Observe that SW is now constrained to be strictly less than N-RW, instead of the earlier less than or equal to N-RW constraint. For the typical case of $\text{MaxDelay}_1 = \text{MaxDelay}_2 =$ MaxDelay, the bound in Theorem 5 simplifies to $\delta \geq \frac{\text{MaxDelay}}{\text{N-SW-RW}}$. This δ formula has been applied to the original Stenning's protocol and to the Arpanet's TCP in Section 1.3.

Bounds on data transfer times

In this implementation, P_1 having a negligible reaction time is modeled by the following equation (instead of R_5):

R_{10} $f_n = \max(a_{n\text{-SW}}, f_{n\text{-}1}+\delta)$

The best case $\{a_n\}$ satisfies R_{10} and $f_n=l_n=a_n$ (which is R_0 with equality). The solution to this is $s_n = n\times\delta$. The worst case $\{a_n\}$ satisfies R_{10} and $a_n = \max(s_n, a_{n\text{-}1}) + T$ (which is R_7 with equality). The solution to this is easily checked to be $a_n = n\times\max(T, \delta) + T$. These observations are summarized in the following:

Theorem 6. This protocol implementation achieves the following performance:

$a_n = n\times\delta$ in the best case

$a_n = n\times\max(T, \delta) + T$ in the worst case

Note that δ must satisfy the bound in Theorem 5. Thus, we see that the bounds in Theorem 6 are worse than the bounds in Theorems 1 and 2, which are achieved by the first two implementations. This is to be expected, because we are using a single timer to achieve 2N-RW-1 time constraints.

Note that if N is sufficiently large so that $\max(\frac{\text{MaxDelay}_1}{\text{N-RW-SW}}, \frac{\text{MaxDelay}_2}{\text{N-1-SW}})$ is less than the minimum time in which the hardware can transmit a message, then the suboptimality of this implementation is of no practical importance. Recall that the optimal bounds in Theorems 1 and 2 assume zero reaction times at the entities, which amounts to assuming that the hardware does not restrict the performance of the entities.

6. LIVENESS PROPERTIES OF THE PROTOCOLS

For the above protocols, we would like to prove that once a data block is accepted from the source at P_1, then it will be acknowledged eventually, provided that the channels eventually deliver messages that are transmitted repeatedly. Additional inference rules are needed to verify such liveness properties [11, 15].

6.1 Inference rules

A liveness property of the protocol system states relationships that values of the system variables eventually satisfy; e.g., the value of state variable s eventually exceeds n for any integer n. A liveness property is a property of the paths in the reachability graph, unlike a safety property which is a property of the nodes in the reachability graph (see Section 3.3). We will verify liveness properties by specifying and verifying inductive properties of *bounded-length* paths in the reachability graph [14, 15]. We assume that any implementation of the protocol system is *fair*, by which we mean the following: any event that is enabled infinitely often will eventually occur.

Given assertions A(**v**) and B(**v**), we say that A(**v**) *leads-to* B(**v**) if for every reachable system state g_0 where $A(g_0)$=True, the following holds: for every unbounded-length path $g_0, g_1, g_2, \ldots$ in the reachability graph, either there is a state g_n for some $n \geq 0$ where $B(g_n)$=True, or the path $g_0, g_1, g_2, \ldots$, is unfair: i.e., there is an event e which is enabled at an infinite number of states in the path but never occurs. If (A(**v**) leads-to B(**v**)) is true, then any fair implementation which is currently in a state that satisfies A(**v**) will eventually be in a state that satisfies B(**v**).

For stating most liveness properties, it necessary to relate values of state variables when A(**v**) holds to their values some time later when B(**v**) holds. For example, we will need to make statements such as (for every integer m)[$s > a = m$ leads-to $s \geq a > m$]. For this purpose, we need to consider assertions whose free variables now include variables different from **v**. Let **m** denote the set of these new variables. Then, A(**v**,**m**) leads-to B(**v**,**m**) means that A(**v**,**m**) leads-to B(**v**,**m**) for each possible value of **m**. For convenience, we will assume each variable in **m** takes nonnegative integer values.

The leads-to construct is similar to the "eventually" operator of temporal logic [11]. We now list a few rather obvious properties of the leads-to relationship.

Leads-to Rule 1. If (A(**v**,**m**) ⇒ B(**v**,**m**)), then (A(**v**,**m**) leads-to B(**v**,**m**)).

Leads-to Rule 2. If (A(**v**,**m**) leads-to (B(**v**,**m**) **or** C(**v**,**m**)) and (C(**v**,**m**) leads-to D(**v**,**m**)), then (A(**v**,**m**) leads-to (B(**v**,**m**)) **or** D(**v**,**m**))

Leads-to Rule 3. If (A(**v**,**m**) leads-to B(**v**,**m**)) and (C(**v**,**m**) leads-to D(**v**,**m**)), then ((A(**v**,**m**) **or** C(**v**,**m**)) leads-to (B(**v**,**m**) **or** D(**v**,**m**))).

With the above rules, we can infer leads-to statements from other leads-to statements. We next consider how leads-to properties can be inferred from the system specifications.

Inference Rule for leads-to. If I(**v**) is invariant and assertions A(**v**,**m**) and B(**v**,**m**) satisfy:

(i) for some event e_0:

(I(**v**) **and** A(**v**,**m**) **and** not B(**v**,**m**)) ⇒ (for some **v**")[e_0(**v**;**v**") **and** B(**v**",**m**)]

(ii) for every event e in the system:

(I(v) **and** A(v,m) **and** not B(v,m) **and** e(v;v")) ⇒ (A(v",m) **or** B(v",m))

then we can infer A(v,m) leads-to B(v,m) via e_0.

The validity of this rule is obvious. Because I(v) is invariant, every reachable state satisfies I(v). Thus, part (i) of the rule ensures that event e_0 is enabled at every reachable state g where A(g,m)=True and B(g,m)=False; the occurrence of e_0 takes the system to a state h where B(h,m)=True. Part (ii) ensures that for every event, if the event is enabled then its occurrence takes the system to a state h where either A(h,m)=True or B(h,m)=True. Thus, in any fair implementation, the system will eventually reach a state where B(v,m) holds. Notice that A(v,m) leads-to B(v,m) via e is a special case of A(v,m) leads-to B(v,m): for every fair unbounded-length path $g_0, g_1, g_2, \ldots$, starting from a state g_0 where (A(g_0,m) **and** not B(g_0,m))=True, there is a state g_n for some $n \geq 1$ such that B(g_n,m)=True and (A(g_j,m) **and** not B(g_j,m))=False for every $j < n$.

As in the case of the inference rule for safety, we can replace I(v) by (I(v) **and** I(v")) in parts (i) and (ii). All the leads-to rules stated earlier can be applied to leads-to-via statements.

Our next inference rule allows us to apply mathematical induction to leads-to statements. Recall that **m** = $(m_1, m_2, \ldots, m_l)$ for some $l \geq 1$, where each m_i can range over the set of non-negative integers. Each permutation $(m_{i_1}, m_{i_2}, \ldots, m_{i_l})$ of the variables in **m** defines a unique lexicographic ordering of the values of **m**: Specifically, if **a** = $(a_1, a_2, \ldots, a_l)$ and **b** = $(b_1, b_2, \ldots, b_l)$ are two different values of **m**, then **a**<**b** if, for some $1 \leq k \leq l$, $a_{i_j} = b_{i_j}$ for all $1 \leq j < k$ and $a_{i_k} < b_{i_k}$.

Induction Rule for leads-to. Given assertions A(v), B(v), D(v,m), and a lexicographic ordering (i.e., some permutation of the variables in **m**) such that:

(A(v) **and** D(v,m)) leads-to (B(v) **or** (A(v) **and** (for some **n**)[D(v,n) **and** **n**>**m**]))

then we can infer (A(v) **and** D(v,**0**)) leads-to (B(v) **or** (for some **n**)[D(v,**n**) **and** **n**>**m**])

The first leads-to statement in the induction rule is referred to as an *inductive leads-to* statement. The induction rule merely applies mathmematical induction to inductive leads-to statements.

In practice, it is quite convenient and sufficient to assume that D(v,m) has the following monotonicity property: if D holds for some value of **m**, then it holds for all lower values of **m** (e.g. D(v,m) ≡ $s \geq m_0$). Then, the above induction rule reduces to the following special case:

Special case of Induction Rule. Given assertions A(v), B(v), D(v,m), and a lexicographic ordering (i.e., some permutation of the variables in **m**) such that:

(i) $D(v,m) \Rightarrow$ (for every n)$[n<m \Rightarrow D(v,n)]$

(ii) ($A(v)$ **and** $D(v,m)$) leads-to ($B(v)$ **or** ($A(v)$ **and** (for some n)[$D(v,n)$ **and** $n>m$]))

then we can infer ($A(v)$ **and** $D(v,0)$) leads-to ($B(v)$ **or** $D(v,m)$)

6.2 Liveness verification of the protocols

We will verify that the protocols satisfy the following liveness property: For any integer n, if a=n and s>a, then eventually Source[n] will be acknowledged and a will exceed n, provided P_1 does not continuosly avoid retransmitting Source[n] while it is outstanding, C_1 does not continuously lose (D,data,ns,uns) messages for any uns in [n..s-1], and C_2 does not continuously lose (ACK,nr,unr) for any unr in [n+1..s]. By induction, this liveness property implies that every outstanding data block is acknowledged eventually. For each of the three protocol implementations, this also implies that P_1 will eventually be ready to accept Source[n+1]. (In fact, in each of the implementations, P_1 will be ready within a *bounded* time.)

To formally state this liveness property, we define the following auxiliary variables:

DCount : array [0..∞] of integers; {For the duration that Source[n] is outstanding, DCount[n] indicates the number of occurrences of Send_D since its last occurrence in which Source[n] was sent. Initially, DCount[0..∞]=0}

$LCount_1$: array [0..∞] of integers; {For the duration that Source[n] is outstanding at P_1, $LCount_1[n]$ indicates the number of times that (D,Source[n],n mod N,n) has been lost in C_1 since its last reception at P_2. Initially, $LCount_1[0..\infty] = 0$}

$LCount_2$: array [0..∞] of integers; {For the duration that Source[n-1] is outstanding at P_1, $LCount_2[n]$ indicates the number of times that (ACK,n mod N,n) has been lost in C_2. Note that Source[n-1] is no longer outstanding as soon as (ACK,n mod N,n) is received at P_1. Initially, $LCount_2[0..\infty] = 0$}

Let $LC_1[n]$ denote ($LCount_1[n]+LCount_1[n+1]+...+LCount_1[s-1]$), and let $LC_2[n]$ denote ($LCount_2[n+1]+LCount_2[n+2]+...+LCount_2[s]$). The desired liveness property can be stated by

L_0 $s>a=n$ leads-to ($s \geq a>n$ **or** $DCount[n] \geq m_0$ **or** $LC_1[n] \geq m_1$ **or** $LC_2[n] \geq m_2$)

We first prove that Source[n] will be received at P_2 unless either P_1 stops retransmitting Source[n], or C_1 continuously loses (D,Source[n],n mod N,n) messages. This property is formally stated as follows:

L_1 $s>r=a=n$ leads-to ($s \geq r>a=n$ **or** $DCount[n] \geq m_0$ **or** $LCount_1[n] \geq m_1$)

Proof of L_1

In the proof, $<n>$ denotes (D,Source[n],n mod N,n). The following leads-to-via statements satisfy the inference rule for leads-to:

(i) M_0 leads-to (M_1 **or** M_2 **or** M_3) via Send_D, where
$M_0 \equiv$ (s>r=a=n **and** $LCount_1[n] \geq m_1$ **and** $DCount[n] \geq m_0$)
$M_1 \equiv$ (s≥r>a=n)
$M_2 \equiv$ (s>r=a=n **and** $LCount_1[n] \geq m_1$ **and** $DCount[n] \geq m_0+1$)
$M_3 \equiv$ (s>r=a=n **and** $LCount_1[n] \geq m_1$ **and** $<n>$ in $\mathbf{z}_1$)
For part (i) of the inference rule, we have: $M_0 \Rightarrow$ (Send_D **and** (M_2'' **or** M_3''))
For part (ii) of the inference rule, we have the following:
(M_0 **and** Send_D) $\Rightarrow$ (M_2'' **or** M_3'')
(M_0 **and** A_1'' **and** Rec_ACK) $\Rightarrow M_0''$ (A_1 is invariant property from Table 1)
(M_0 **and** Rec_D **and** A_1'') $\Rightarrow$ (M_1'' **or** M_0'')
(M_0 **and** e) $\Rightarrow M_0''$ for every other event e
Property (i) follows from leads-to inference rule; let $A \equiv M_0$ and $B \equiv$(M_1 **or** M_2 **or** M_3).

(ii) M_3 leads-to (M_1 **or** M_4) via loss event of C_1, where
$M_4 \equiv$ (s>r=a=n **and** $LCount_1[n] \geq m_1+1$)
For part (i) of the inference rule, we have: $M_3 \Rightarrow$ (Losing $<n>$ in C_1 **and** M_4'')
For part (ii) of the inference rule, we have the following:
(M_3 **and** A_1'' **and** Rec_ACK) $\Rightarrow M_3''$
(M_3 **and** Rec_D **and** A_1'') $\Rightarrow$ (M_1'' **or** M_3'')
(M_3 **and** Loss event of C_1) $\Rightarrow$ (M_3'' **or** M_4'')
(M_3 **and** e) $\Rightarrow M_3''$ for every other event e
Property (ii) follows by letting $A \equiv M_3$ and $B \equiv$(M_1 **or** M_4) in inference rule.

Applying leads-to rule 2 to (i) and (ii), we have (let M_0, (M_1 **or** M_2), M_3, M_4 be A, B, C, and D respectively):

(iii) M_0 leads-to (M_1 **or** M_2 **or** (M_1 **or** M_4))
which can be rewritten as
(iv) (s>r=a=n **and** $DCount[n] \geq m_0$ **and** $LCount_1[n] \geq m_1$)
leads-to (s≥r>a=n **or** (s>r=a=n **and**
($LCount_1[n] \geq m_1+1$ **or** ($LCount_1[n] \geq m_1$ **and** $DCount[n] \geq m_0+1$))))

We can infer L_1 from (iv) by applying the leads-to induction rule: Use the lexicographic ordering due to the permutation $\mathbf{m}=(m_1,m_0)$, let $D(\mathbf{v},\mathbf{m}) \equiv$ ($LCount_1[n] \geq m_1$ **and** $DCount[n] \geq m_0$), let $A(\mathbf{v}) \equiv$(s≥r>a=n), and let $B(\mathbf{v}) \equiv$(s>r=a=n).
End of proof of L_1

We next prove that once P_2 has received Source[n], then P_1 eventually receives an acknowledgement (ACK,nr,unr) where n<unr≤s, unless either C_2 continuously loses such (ACK,nr,unr) messages for any unr in [n+1..s], or C_1 continuously loses (D,Source[uns],ns,uns)

messages for any uns in [n..s-1] (this can cause P_2 to not send ACK messages repeatedly). This property is formally stated as follows:

L_2 $s \geq r > a = n$ leads-to ($s \geq r \geq a > n$ **or** $LC_1[n] \geq m_1$ **or** $LC_2[n] \geq m_2$)

Proof of L_2

In the proof, <uns> denotes (D,Source[uns],ns,uns) and <unr> denotes (ACK,nr,unr). The following leads-to-via statements satisfy the inference rule for leads-to:

(i) M_0 leads-to (M_1 **or** M_2 **or** M_3) via Send_D, where
$M_0 \equiv$ ($s \geq r > a = n$ **and** $LC_1[n] \geq m_1$ **and** $LC_2[n] \geq m_2$)
$M_1 \equiv$ ($s \geq r \geq a > n$)
$M_2 \equiv$ ($s \geq r > a = n$ **and** $LC_1[n] \geq m_1$ **and** $LC_2[n] \geq m_2$
and (for some uns in [n..s-1])[<uns> in $\mathbf{z}_1$])
$M_3 \equiv$ ($s \geq r > a = n$ **and** $LC_2[n] \geq m_2$ **and** SendACK=True)
For part (i) of the inference rule, we have: $M_0 \Rightarrow$ (Send_D **and** M_2")
For part (ii) of the inference rule, we have the following:
(M_0 **and** Send_D) $\Rightarrow M_2$"
(M_0 **and** A_1" **and** Rec_ACK) $\Rightarrow$ (M_0" **or** M_1")
(M_0 **and** Rec_D **and** A_1") $\Rightarrow M_3$"
(M_0 **and** e) $\Rightarrow M_0$" for every other event e
Property (i) follows by letting $A \equiv M_0$ and $B \equiv$(M_1 **or** M_2 **or** M_3) in inference rule.

(ii) M_2 leads-to (M_1 **or** M_3 **or** M_4) via Loss event of C_1, where
$M_4 \equiv$ ($s \geq r > a = n$ **and** $LC_1[n] \geq m_1+1$ **and** $LC_2 \geq m_2$)
For part (i) of the inference rule, we have: $M_2 \Rightarrow$ (Losing <uns> in C_1 **and** M_4")
For part (ii) of the inference rule, we have the following:
(M_2 **and** A_1" **and** Rec_ACK) $\Rightarrow$ (M_2" **or** M_1")
(M_2 **and** Rec_D **and** A_1") $\Rightarrow M_3$"
(M_2 **and** Loss event of C_1) $\Rightarrow$ (M_2" **or** M_4")
(M_2 **and** e) $\Rightarrow M_2$" for every other event e
Property (ii) follows by letting $A \equiv M_2$ and $B \equiv$(M_1 **or** M_3 **or** M_4) in inference rule.

(iii) M_3 leads-to (M_1 **or** M_5) via Send_ACK, where
M_5 ($s \geq r > a = n$ **and** $LC_2[n] \geq m_2$ **and** (for some unr in [n+1..s])[<unr> in $\mathbf{z}_2$])
For part (i) of the inference rule, we have: $M_3 \Rightarrow$ (Send_ACK **and** M_5")
For part (ii) of the inference rule, we have the following:
(M_3 **and** A_1" **and** Rec_ACK) $\Rightarrow$ (M_3" **or** M_1")
(M_3 **and** Send_ACK) $\Rightarrow M_5$"
(M_3 **and** e) $\Rightarrow M_3$" for every other event e
Property (iii) follows by letting $A \equiv M_3$ and $B \equiv$(M_1 **or** M_5) in inference rule.

(iv) M_5 leads-to (M_1 **or** M_6) via Loss event of C_2, where

$M_6 \equiv (s \geq r > a = n$ **and** $LC_2[n] \geq m_2+1)$

For part (i) of the inference rule, we have: $M_5 \Rightarrow$ (Losing <unr> in C_2 **and** M_6")

For part (ii) of the inference rule, we have the following:

(M_5 **and** A_1" **and** Rec_ACK) $\Rightarrow$ (M_5" **or** M_1")

(M_5 **and** Loss event of C_2) $\Rightarrow$ (M_5" **or** M_6")

(M_5 **and** e) $\Rightarrow$ M_5"   for every other event e

Property (iv) follows by letting $A \equiv M_5$ and $B \equiv (M_1$ **or** $M_6)$ in inference rule.

Note that (M_5 leads-to (M_1 **or** M_6) via Rec_ACK) also holds.

Applying the leads-to rule 2 to (i) and (ii), we get (M_0 leads-to (M_1 **or** M_3 **or** M_4)). Applying the leads-to rule 2 to this and (iii), we get (M_0 leads-to (M_1 **or** M_4 **or** M_5)). Applying the leads-to rule 2 to this and (iv), we get (M_0 leads-to (M_1 **or** M_4 **or** M_6)), which can be written as

(v) $(s \geq r > a = n$ **and** $LC_1[n] \geq m_1$ **and** $LC_2[n] \geq m_2)$ leads-to
$(s \geq r \geq a > n$ **or** $(s \geq r > a = n$ **and**
$((LC_1[n] \geq m_1+1$ **and** $LC_2[n] \geq m_2)$ **or** $LC_2[n] \geq m_2+1))$

We can infer L_2 from (v) by using the leads-to induction rule: Use the permutation $\mathbf{m}=(m_2,m_1)$ and let $D(\mathbf{v},\mathbf{m}) \equiv (LC_2[n] \geq m_2$ **and** $LC_1[n] \geq m_1)$, let $A(\mathbf{v}) \equiv (s \geq r \geq a > n)$, and let $B(\mathbf{v}) \equiv (s \geq r > a = n)$.

End of proof of L_2

The two liveness properties L_1 and L_2 together imply L_0.

Proof of L_0

By leads-to rules 1 and 3, we can **or** both sides of L_1 by $(s \geq r > a = n)$ to obtain

(i) $(s \geq r \geq a$ **and** $s > a = n)$ leads-to
$(s \geq r > a = n$ **or** $DCount[n] \geq m_0$ **or** $LCount_1[n] \geq m_1)$

Applying leads-to rule 2 to (i) and L_2, we obtain

(ii) $(s \geq r \geq a$ **and** $s > a = n)$ leads-to
$(s \geq r \geq a > n$ **or** $LC_1[n] \geq q_1$ **or** $LC_2[n] \geq q_2$ **or** $DCount \geq m_0$ **or** $LCount_1[n] \geq m_1)$

Property (ii) implies L_0 because $(LCount_1[n] \geq m_1 \Rightarrow LC_1[n] \geq m_1)$ and $s \geq r \geq a$ is invariant (from safety property A_1).

End of proof of L_0

7. PROTOCOLS WITH REAL-TIME PROGRESS

We now modify the above protocols so that unacknowledged data blocks are acknowledged within a bounded response time T, provided that the channels do not consistently perform badly. Such a real-time progress property is more realistic than the liveness property proved above in Section 6. In practice, if progress is not achieved within time T, then P_1 aborts the connection with P_2.

Let Delay_i ($\leq \text{MaxDelay}_i$) be the delay that a message is *expected* to encounter in channel C_i. We say that a message m in C_i is *overdelayed* if it is not received within Delay_i time of its send. ($\text{Delay}_i \ll \text{MaxDelay}_i$ for a realistic channel.) Note that if $\text{Delay}_i = \text{MaxDelay}_i$ then overdelaying message m corresponds to losing m and any of its duplicates.

Entity P_2 will now send an ACK message within a specified MaxResponseTime of receiving a D message.

Define RoundTripDelay = ($\text{Delay}_1 + \text{Delay}_2$ + MaxResponseTime). For any n, whenever s>a=n holds, then entity P_1 will do a *test* transmission of Source[n] once every RTripDelay local time units, where RTripDelay = $(1+\epsilon_1) \times$ RoundTripDelay. By thus separating successive test transmissions, we make sure that different tests do not fail due to the same channel failure.

We will prove that the following worst-case progress property holds for any $n \geq 0$ and for any MaxFailCount ≥ 0: $s \geq a > n$ holds within T (= MaxFailCount×RTripDelay) seconds of s>a=n first holding, provided that [the number of times that C_1 has overdelayed a D message containing Source[n]] + [the number of times that C_2 has overdelayed an ACK message acknowledging Source[n]] does not exceed MaxFailCount.

To formally state this property, we define the following auxiliary variables:

OutTimer : (Off,0,1,...); {Auxiliary local timer that indicates the local time elapsed since s>a=n became true. OutTimer=Off if s=a. Initially, OutTimer=Off}

SCount_1 : array[0..∞] of integers; {SCount_1[n] indicates the number of times that C_1 overdelays a test message (D,data,ns,n) while s>a=n holds. Initially, $\text{SCount}_1[0..\infty]=0$}

SCount_2 : array[0..∞] of integers; {SCount_2[n] indicates the number of times that C_2 overdelays an (ACK,nr,n) message while $s \geq n > a$ holds. Initially, $\text{SCount}_2[0..\infty]=0$}

Let $SC_2[a+1]$ denote $\text{SCount}_2[a+1] + \text{SCount}_2[a+2] + \ldots + \text{SCount}_2[r]$. The worst-case progress property can be formally stated as follows:

D_0 $\quad \text{OutTimer} > \text{MaxFailCount} \times \text{RTripDelay} \Rightarrow \text{SCount}_1[a] + SC_2[a+1] \geq \text{MaxFailCount}$

We will establish the invariance of D_0. Note that D_0 is a safety property, and not liveness property requiring the leads-to operator. (The worst-case progress property can also be stated formally in terms of the leads-to operator.)

7.1 Modified protocol implementation

We now modify the protocol implementations of Section 5 so that they enforce the required real-time behavior. Auxiliary variables needed to verify D_0 are also specified.

There is now a local time event of accurracy ϵ_2 at P_2. Let MResponseTime = $(1-\epsilon_2)\times$MaxResponseTime - 1. Include the following at P_2:

SendACKTimer : (Off,0,1,...,MResponseTime); {Local timer that indicates the elapsed time following the earliest reception of a D message for which an ACK response has not yet been sent. SendACKTimer=Off when there is no such D message. Initially, SendACKTimer=Off}

SendACKTimerG : (Off,0,1,...); {Auxiliary ideal timer which tracks SendACKTimer. Initially, SendACKTimerG=Off}

Both SendACKTimer and SendACKTimerG are reset to Off in the Send_ACK event, and reset to 0, if it was Off, in the Rec_D event. SendACKTimer is constrained by the timer axiom: SendACKTimer$\neq$Off $\Rightarrow$ SendACKTimer $<$ MResponseTime

We have the following modifications at P_1.

TestTimer : (Off,0,1,...,RTripDelay); {Local timer. If s>a=n and no test transmission of Source[n] has yet occurred, then TestTimer indicates the local time elapsed since s>a=n became true. If s>a=n and at least one test transmission of Source[n] has occurred, then TestTimer indicates the local time elapsed following the last test transmission of Source[n]. If s=a then TestTimer=Off. Initially, TestTimer=Off}

TestTimerG : (Off,0,1,...); {Auxiliary ideal timer which tracks TestTimer. Initially, TestTimerG=Off}

TestCount : integer; {Indicates the number of test transmissions of Source[a] that have already occurred. Initially, TestCount=0}

A test transmission of Source[a] is done whenever s>a and TestTimer=RTripDelay; the test transmission increments TestCount by 1 and resets TestTimer to 0. P_1 enforces the following timer axiom: TestTimer$\neq$Off $\Rightarrow$ TestTimer $<$ RTripDelay

In order to distinguish between different test transmissions of a data block Source[n], we need the following:

tn : integer; {tn is an auxiliary field present in each D message. tn equals the (updated) value of TestCount if the D message was sent in a test transmission. tn=0 if and only if the D message was sent in a non test transmission}

In order to update $SCount_1$ and $SCount_2$, we define the following auxiliary variables:

TnRecd : array[0..∞] of integer sequence; {TnRecd[uns] indicates the tn values in the (D,data,ns,uns,tn) messages received at P_2. Initially, TnRecd[0..∞]=null}

ACKSent : array[0..∞] of integer sequence; {ACKSent[unr] is a sequence of ideal time values; for

each transmission of an (ACK,nr,unr) message, there is an entry indicating the ideal time elapsed since its transmission. Initially, ACKSent[0..∞]=null}

The events of the modified protocol are given in Tables 7, 8, and 9. Each event is a refinement of some event in Tables 2-6; Send_Test_D is a refinement of the former Send_D. Thus, the safety properties of Table 1 continue to hold for this modified protocol.

7.2 Real-time progress verification

We first observe that the following property, which involves P_1 alone, is invariant (proof is trivial):

D_1 $\quad$ s>a $\Rightarrow$ (OutTimer = TestCount$\times$RTripDelay + TestTimer
and ((TestTimer, TestTimerG) started at 0))

Suppose that s>a=n holds. Just before a test transmission, we expect TestCount $\leq$ $SCount_1$[n] + SC_2[n+1] to hold. Therefore, just after the test transmission, we have TestCount $\leq$ $SCount_1$[n] + SC_2[n+1] + 1. Consider the following evolution for the test:

(a) The test message (or any duplicate of it) is received at P_2 before TestTimerG exceeds $Delay_1$.
(b) The responding ACK message, which acknowledges Source[n], is sent by P_2 before TestTimerG exceeds $Delay_1$ + MaxResponseTime.
(c) The ACK message is received at P_1 (and Source[n] is acknowledged) before TestTimerG exceeds RoundTripDelay.

In this case, TestCount $\leq$ $SCount_1$[n] + SC_2[n+1] + 1 holds during the entire course of the test. Any deviation from the above course causes TestCount $\leq$ $SCount_1$[n] + SC_2[n+1] to hold when TestTimerG exceeds RoundTripDelay. If event(a) does not occur then C_1 has overdelayed Source[n]. If event(a) occurs then event(b) will occur because of the timer axiom constraint; if event(c) does not occur then C_2 has overdelayed an acknowledgement to Source[n]. Thus, the following is invariant (formal proof in Appendix C):

D_2 $\quad$ TestCount$\geq$1 $\Rightarrow$ ($D_{2.1}$ **or** $D_{2.2}$ **or** $D_{2.3}$ **or** $D_{2.4}$)
where

$D_{2.1}$ TestTimerG in [0..$Delay_1$] **and** TestCount not in TnRecd[a]
and TestCount $\leq$ $SCount_1$[a] + SC_2[a+1] + 1

$D_{2.2}$ (for some rectime in [0..$Delay_1$])
[TestTimerG in [rectime..rectime+MaxResponseTime]
and r>a **and** SendACK=True **and** SendACKTimerG $\geq$ TestTimerG - rectime
and ((SendACKTimerG, SendACKTimer) started at 0)
and TestCount $\leq$ $SCount_1$[a] + SC_2[a+1] + 1]

$D_{2.3}$ (for some sendtime in [0..$Delay_1$+MaxResponseTime])

[TestTimerG in [sendtime..sendtime+$Delay_2$]
and (for some n in [a+1..r])[(TestTimerG - sendtime) in ACKSent[n]]
and TestCount $\leq$ $SCount_1$[a] + SC_2[a+1] + 1]

$D_{2.4}$ TestTimerG $\geq$ min($Delay_1$, $Delay_2$) + 1
and TestCount $\leq$ $SCount_1$[a] + SC_2[a+1]

D_0 is implied by (D_1 **and** D_2) as follows: D_2 implies that (TestCount $\leq$ $SCount_1$[a] + SC_2[a+1] + 1) is invariant. (D_1 **and** timer axiom for TestTimer) implies the invariance of (for all m$\geq$0)[OutTimer > m$\times$RTripDelay $\Rightarrow$ TestCount $\geq$ m]. The above two imply D_0.

REFERENCES

[1] Bartlett, K. A. et al, "A note on reliable full-duplex transmission over half-duplex links," *Commun. of the ACM,* May 1980.

[2] Brand, D. and W. H. Joyner, "Verification of HDLC," *IEEE Trans. Commun.*, Vol. 30, 5, May 1982.

[3] CCITT, Draft revised CCITT reccommendation X.25, February 1980.

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

[5] DiVito, B. L., "Mechanical verification of a data transport protocol," *Proc. ACM SIGCOMM '83,* Austin, Texas, pp. 30-37, March 1983.

[6] Hailpern, B. T. and S. S. Owicki, "Modular verification of computer communication protocols," *IEEE Trans. on Commun.*, COM-31, 1, January 1983.

[7] International Standards Organization, "Data Communication—High-level Data Link Control Procedures—Frame Structure," Ref. No. ISO 3309, Second Edition, 1979. "Data Communications—HDLC Procedures—Elements of Procedures," Ref. No. ISO 4335, First Edition, 1979. International Standards Organization, Geneva, Switzerland.

[8] Lamport, L, "Time, Clocks, and the Ordering of Events in a Distributed System," *Comm. ACM,* Vol. 21, No. 7, July 1978, pp. 558-565.

[9] Lam, S. S. and A. U. Shankar, "Protocol Verification via Projections," *IEEE Trans. on Software Eng.,* Vol. SE-10, No. 4, July 1984, pp. 325-342.

[10] Misra, J. and K. M. Chandy, "Proofs of Networks of Processes," *IEEE Trans. Soft. Eng.*, Vol. SE-7, No. 4, July 1981.

[11] Owicki, S. and L. Lamport, "Proving Liveness Properties of Concurrent Programs," *ACM TOPLAS*, Vol. 4, No. 3, July 1982, pp. 455-495.

[12] Postel, J. (ed.), "DOD Standard Transmission Control Protocol," Defense Advanced Research Projects Agency, Information Processing Techniques Office, RFC 761, IEN 129, January 1980; in *ACM Computer Communication Review*, Vol. 10, No. 4, October 1980, pp. 52-132.

[13] Shankar, A. U. and S. S. Lam, "An HDLC Protocol Specification and its Verification Using Image Protocols," *ACM Trans. on Computer Systems*, Vol. 1, No. 4, November 1983, pp. 331-368.

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

[15] Shankar, A. U. and S. S. Lam, "Time-dependent distributed systems: proving safety, liveness and real-time properties," Tech. Rep. CS-TR-1586, Computer Science Dept., Univ. of Maryland, also TR-85-24, Computer Science Dept., Univ. of Texas, October 1985, (submitted to ACM TOPLAS).

[16] Sloan, L., "Mechanisms that Enforce Bounds on Packet Lifetimes," *ACM Trans. Comput. Syst.*, Vol. 1, No. 4, Nov. 1983, pp. 311-330.

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

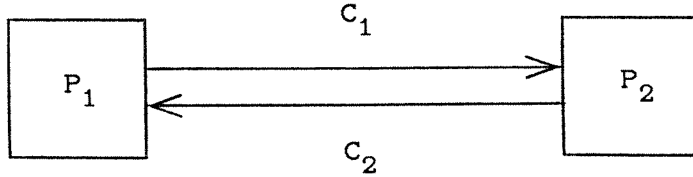

Figure 1. The protocol system.

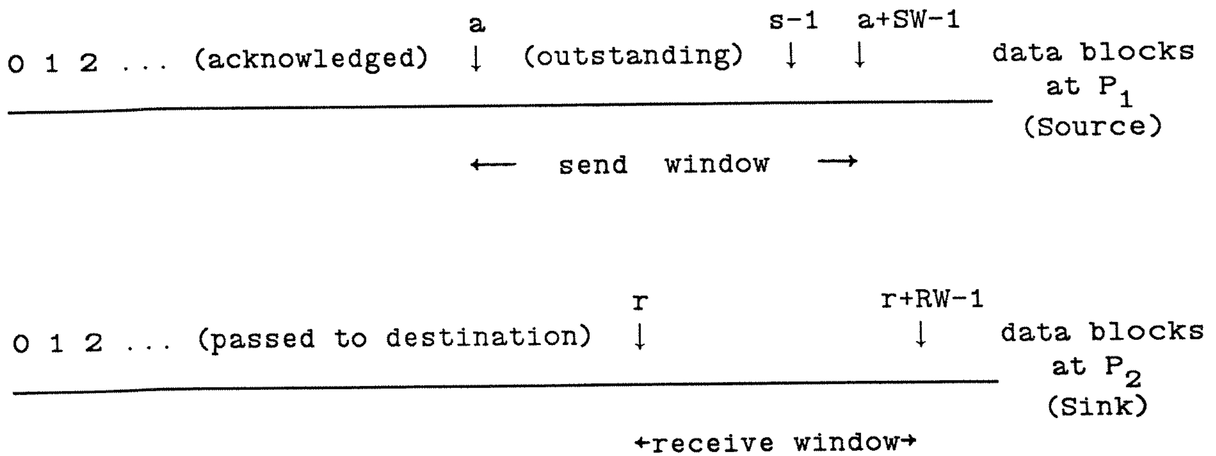

Figure 2. The send and receive windows.

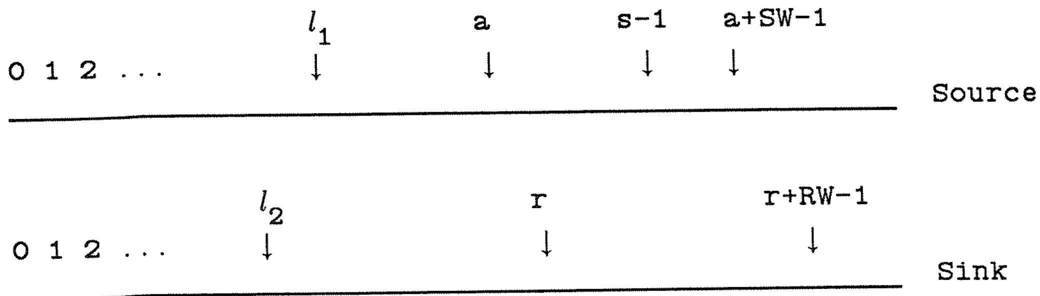

Figure 3. An illustration of l_1, the smallest uns value in C_1, and l_2, the smallest unr value in C_2.

Table 1: Safety properties of the protocol

{Properties relating Source, Sink, s, a, r, vs, va, vr, N, SW, RW}

A_0 (for all n in [0..r-1])[Sink[n]=Source[n]]
A_1 $0 \leq a \leq r \leq s$
B_{11} $s \leq a+SW$
B_5 $1 \leq RW \leq N-1$
B_{10} $1 \leq SW \leq N-RW$
A_2 (for all n in [r+1..r+RW-1])[Sink[n]≠empty ⇒ Sink[n]=Source[n]]
A_3 (for all n in [min(s, r+RW)..∞])[Sink[n]=empty]
B_0 Sink[r]=empty
B_1 vs=s mod N **and** va=a mod N **and** vr=r mod N

{Properties relating D messages, Source, DTimeG, Sink, s}

B_2 (D,data,ns,uns) in $\mathbf{z}_1$ ⇒ (uns in [0..s-1] **and** data=Source[uns] **and** ns=uns mod N)
B_6 <(D,data,ns,uns), age> in $\mathbf{z}_1$ ⇒ (age ≥ DTimeG[uns] ≥ 0)
A_9 (D,data,ns,uns) in $\mathbf{z}_1$ ⇒ uns in [s-N+RW..s-1]
A_{11} (for all n in [0..s-N+RW-1])[DTimeG[n] > $MaxDelay_1$]

{Properties relating ACK messages, r, s, RTimeG}

B_4 (ACK,nr,unr) in $\mathbf{z}_2$ ⇒ (unr in [0..r] **and** nr = unr mod N)
B_8 <(ACK,nr,unr), age> in $\mathbf{z}_1$ ⇒
((unr<r ⇒ age≥RTimeG[unr]) **and** (unr≥1 ⇒ age≤RTimeG[unr-1]))
A_8 (ACK,nr,unr) in $\mathbf{z}_2$ ⇒ unr in [s-N+1..r]
B_7 RTimeG[0] ≥ RTimeG[1] ≥ ... ≥ RTimeG[r-1] ≥ 0

{Properties relating ATimeG, a, s, RTimeG}

B_9 ATimeG[0] ≥ ATimeG[1] ≥ ... ≥ ATimeG[a-1] ≥ 0
A_{12} (for all n in [0..a-1])[RTimeG[n] ≥ ATimeG[n]]
A_{13} (for all n in [0..s-N])[ATimeG[n] > $MaxDelay_2$]

Table 2: Variables of P_1

Source : array[0..s-1] of DataSet; {Auxiliary variable. Initially, s=0 and Source is the null array}

a : $0..\infty$; {Auxiliary variable initialized to 0}

vs, va : 0..N-1; {Initially, va=vs=0}

DTimeG : array[$0..\infty$] of (Off,0,1,...); {Auxiliary ideal timers. Initially, DTimeG[$0..\infty$]=Off}

DTimer : array[0..N-1] of (Off,0,1,...,$MDelay_1$); {Local timers. Initially, DTimer[0..N-1]=Off}

ATimeG : array[$0..\infty$] of (Off,0,1,...); {Auxiliary ideal timers. Initially, ATimeG[$0..\infty$]=Off}

ATimer : array[0..N-1] of (Off,0,1,...,$MDelay_2$); {Local timers. Initially, ATimer[0..N-1]=Off}

Table 3: Events of P_1

1. AcceptData($\mathbf{v}_1;\mathbf{v}_1''$)
   $\equiv$ (vs$\ominus$va $\leq$ SW-1 **and** DTimer[vs$\oplus$RW]=Off **and** ATimer[vs$\oplus$1]=Off)
   $\rightarrow$ (Source[s]" in DataSet **and** vs"=vs$\oplus$1 **and** s"=s+1)

2. Send_D($\mathbf{v}_1,\mathbf{z}_1;\mathbf{v}_1'',\mathbf{z}_1''$)
   $\equiv$ (for some i in [0..s-a-1])[$Send_1$((D,Source[a+i],va$\oplus$i,a+i), $\mathbf{z}_1;\mathbf{z}_1''$)
   and DTimer[va$\oplus$i]"=0 **and** DTimeG[a+i]"=0]

3. Rec_ACK($\mathbf{v}_1,\mathbf{z}_2;\mathbf{v}_1'',\mathbf{z}_1''$)
   $\equiv$ (for some nr in [0..N-1])(for some integer unr)
   [Rec_2($\mathbf{z}_2$;(ACK,nr,unr),$\mathbf{z}_2''$)
   and (1 $\leq$ nr$\ominus$va $\leq$ vs$\ominus$va $\rightarrow$ (va"=nr **and** a" = a + nr$\ominus$va
   and (for all i in [0..nr$\ominus$va-1])[ATimer[va$\oplus$i]"=0 **and** ATimerG[a+i]"=0]))]

Table 4: Variables of P_2

Sink : array[0..∞] of DataSet $\cup$ {empty}; {Auxiliary variable. Initially, Sink[0..∞]=empty}

r : 0..∞; {Auxiliary variable. Initially, r=0}

vr : 0..N-1; {Initially, vr=0}

SendACK : boolean; {Initially, SendACK=False}

RTimeG : array[0..∞] of (Off,0,1,...); {Auxiliary ideal timers. Initially, RTimeG[0..∞]=Off}

Table 5: Events of P_2

1. Send_ACK($\mathbf{v}_2,\mathbf{z}_2;\mathbf{v}_2'',\mathbf{z}_2''$)
   $\equiv$ SendACK=True **and** Send_2((ACK,vr,r), $\mathbf{z}_2;\mathbf{z}_2''$) **and** SendACK"=False

2. Rec_D($\mathbf{v}_2,\mathbf{z}_1;\mathbf{v}_2'',\mathbf{z}_2''$)
   $\equiv$ (for some data in DataSet)(for some ns in [0..N-1])(for some integer uns)
   [Rec_1($\mathbf{z}_1$;(D,data,ns,uns),$\mathbf{z}_1''$) **and**
   ((0$\leq$ns$\ominus$vr$\leq$RW-1 **and** Sink[r+ns$\ominus$vr]=empty)
   $\rightarrow$ (Sink[r+ns$\ominus$vr]"=data **and** (ns=vr $\rightarrow$ SinkData($\mathbf{v}_2$; $\mathbf{v}_2''$))))
   and SendACK"=True]

   where SinkData($\mathbf{v}_2$; $\mathbf{v}_2''$)
   $\equiv$ (for some k in [1..RW])[Sink[r+k]=empty
   and (for all j in [1..k-1])[Sink[r+j]$\neq$empty **and** RTimeG[r+j]"=0]
   and r"=r+k **and** vr"=vr$\oplus$k]

Table 6: Time events

Ideal time event

$\equiv \eta" = \eta+1$ **and** $\mathbf{z}_1" = \text{next}(\mathbf{z}_1)$ **and** $\mathbf{z}_2" = \text{next}(\mathbf{z}_2)$
and RTimeG" = next(RTimeG) **and** ATimeG" = next(ATimeG)
and DTimeG" = next(DTimeG)
and $\text{AccuracyAxiom}_1(\eta_1, \eta")$ **and** $\text{TimerAxiom}_1(\mathbf{z}_1")$ **and** $\text{TimerAxiom}_2(\mathbf{z}_2")$

Local time event for P_1

$\equiv \eta_1" = \eta_1+1$ **and** DTimer" = next(DTimer) **and** ATimer" = next(ATimer)
and $\text{AccuracyAxiom}_1(\eta_1", \eta)$

Table 7: Events of P_1 in Real-time protocol

1. $\text{AcceptData}(\mathbf{v}_1; \mathbf{v}_1")$
   $\equiv$ (vs$\ominus$va $\leq$ SW-1 **and** DTimer[vs$\oplus$RW]=Off **and** ATimer[vs$\oplus$1]=Off)
   $\rightarrow$ (Source[s]" in DataSet **and** vs"=vs$\oplus$1 **and** s"=s+1
   and (vs=va $\rightarrow$ TestTimer"=OutTimer"=0))

2. $\text{Send_D}(\mathbf{v}_1, \mathbf{z}_1; \mathbf{v}_1", \mathbf{z}_1")$
   $\equiv$ (for some i in [0..s-a-1])[Send_1((D,Source[a+i],va$\oplus$i,a+i,0), $\mathbf{z}_1; \mathbf{z}_1"$)
   and DTimer[va$\oplus$i]"=0 **and** DTimeG[a+i]"=0]

3. $\text{Send_Test_D}(\mathbf{v}_1, \mathbf{z}_1; \mathbf{v}_1", \mathbf{z}_1")$
   $\equiv$ (vs$\neq$va **and** TestTimer=RTripDelay **and** Send_1((D,Source[a],va,a,TestCount),$\mathbf{z}_1; \mathbf{z}_1"$)
   and DTimer[va]"=DTimeG[a]"=TestTimer"=0)

4. $\text{Rec_ACK}(\mathbf{v}_1, \mathbf{z}_2; \mathbf{v}_1", \mathbf{z}_1")$
   $\equiv$ (for some nr in [0..N-1])(for some integer unr)[Rec_2($\mathbf{z}_2$;(ACK,nr,unr),$\mathbf{z}_2"$)
   and (1 $\leq$ nr$\ominus$va $\leq$ vs$\ominus$va $\rightarrow$ (va"=nr **and** a" = a + nr$\ominus$va
   and ((va"$\neq$vs $\rightarrow$ TestTimer"=OutTimer"=0)
   (va"=vs $\rightarrow$ TestTimer"=OutTimer"=Off))
   and (for all i in [0..nr$\ominus$va-1])[ATimer[va$\oplus$i]"=0 **and** ATimerG[a+i]"=0]))]

Table 8: Events of P_2 in Real-time protocol

1. Send_ACK(v_2,z_2;v_2'',z_2'')
   ≡ SendACK=True **and** Send$_2$((ACK,vr,r), z_2;z_2'')
   and SendACK"=False **and** SendACKTimer"=SendACKTimerG"=Off
   and ACKSent[r]"=(ACKSent[r],0)

2. Rec_D(v_2,z_1;v_2'',z_2'')
   ≡ (for some data in DataSet)(for some ns in [0..N-1])(for some integers uns and tn)
   [Rec$_1$(z_1;(D,data,ns,uns,tn),z_1'')
   and ((0≤ns⊖vr≤RW-1 **and** Sink[r+ns⊖vr]=empty)
   → (Sink[r+ns⊖vr]"=data **and** (ns=vr → SinkData(v_2;v_2''))))
   and SendACK"=True
   and (SendACKTimer≠Off → SendACKTimer"=SendACKTimerG"=0)
   and (TnRecd[uns]"=(TnRecd[uns], tn))]

Table 9: Time Events in Real-time protocol

Ideal time event
≡ (ideal time event predicate from Table 6)
and AccuracyAxiom$_2$(η_2,η") **and** SendACKTimerG"=next(SendACKTimerG)
and TestTimerG"=next(TestTimerG)
and ((TestTimerG=Delay$_1$ **and** TestCount not in TnRecd[a])
→ SCount$_1$[a]"=SCount$_1$[a]+1)
and ACKSent"=next(ACKSent)
and ((for all n in [a+1..s])[Delay$_2$ in ACKSent[n] → SCount$_2$[n]"=SCount$_2$+1]

Local time event for P_1
≡ (local time event predicate from Table 6)
and TestTimer"=next(TestTimer) **and** OutTimer"=next(OutTimer)
and TestTimer" < RTripDelay)

Local time event for P_2
≡ η_2"=η_2+1 **and** AccuracyAxiom$_2$(η_2,η) **and** SendACKTimer"=next(SendACKTimer)
and SendACKTimer" < MResponseTime

Appendix A

Proof of invariance of B_2

Initial Conditions:

(a) B_2 — (z_1=null)

Channel errors, Rec_D:

(a) B_2'' — (B_2, $<m,t>$ in $z_1'' \Rightarrow <m,t>$ in z_1)

Send_D:

(a) i in [0..s-a-1] **and** $z_1''=(z_1,<(D,Source[a+i],va\oplus i,a+i),0>)$ — (Send_D)
(b) B_2'' — (a, B_2, B_1)

AcceptData:

(a) B_2'' — (B_2, s"=s+1)

Other events do not affect B_2.

End of proof

Proof of invariance of B_4.

Initial Conditions:

(a) B_4 — (z_2=null)

Channel errors, Rec_ACK:

(a) B_4'' — (B_4, $<m,t>$ in $z_2'' \Rightarrow <m,t>$ in z_2)

Send_ACK:

(a) B_4'' — (B_4, $z_2''=(z_2,<(ACK,vr,r),0>)$, B_1)

Rec_D:

(a) $r'' \geq r$, $z_2''=z_2$ — (Rec_D)
(b) B_4'' — (a, B_4)

Other events do not affect B_4.

End of proof

Proof of Lemma 3

Initial Conditions:

(a) $A_{0\text{-}3,8\text{-}10}$ — (a=r=s=0, Sink[0..RW-1]=empty, $z_1=z_2$=null)

Time events do not affect $A_{0\text{-}3,8\text{-}10}$.

Channel errors: $A_{0\text{-}3,10}$ not affected.

(a) $A''_{8\text{-}9}$ — ($A_{8\text{-}9}$, (for i=1,2)[$\langle m,t\rangle$ in $\mathbf{z}_i'' \Rightarrow \langle m,t\rangle$ **in** $\mathbf{z}_i$])

Send_D: $A_{0\text{-}3,8,10}$ not affected.

(a) i in [0..s-a-1] **and** $\mathbf{z}_1''=(\mathbf{z}_1, \langle(D,data,ns,a+i),0\rangle)$ — (Send_D)
(b) A_9'' — (a, A_9)

Send_ACK: $A_{0\text{-}3,9\text{-}10}$ not affected.

(a) A_8'' — (A_8, $\mathbf{z}_2''=(\mathbf{z}_2, \langle(ACK,vr,r),0\rangle)$)

Rec_ACK: $A_{0,2\text{-}3,9}$ not affected.

(a) $\mathbf{z}_2=(\langle(ACK,nr,unr),age\rangle, \mathbf{z}_2'')$ — (Rec_ACK)
(b) A_8'' — (a, A8)
(c) unr in [s-N+1..r], nr=unr mod N — (A_8, B_4)

(d1) unr in [s-N+1..a] — (assumption)
(d2) nr⊖va not in [1..vs⊖va] — (d1, c, A_{10}, Lemma 2)
(d3) a''=a — (d2, Rec_ACK)

(d) d1 ⇒ $A_{1,10}''$ — (d3, d1, A_1, A_{10})

(e1) unr in [a+1..r] — (assumption)
(e2) nr⊖va in [1..vs⊖va], unr=a+nr⊖va — (e1, c, A_1, A_{10}, Lemma 2)
(e3) a''=unr — (e2, Rec_ACK)

(e) e1 ⇒ $A_{1,10}''$ — (e3, e2, e1, $A_{1,10}$)
(f) $A_{1,10}''$ — (c, d, e)

Rec_D: A_{10} not affected.

(a) $\mathbf{z}_1=(\langle(D,data,ns,uns),age\rangle)$ — (Rec_D)
(b) A_9'' — (a, A_9)
(c) data=Source[uns], ns=uns mod N — (B_2)
(d) uns in [s-N+RW..s-1] — (A_9)

(e1) uns in [s-N+RW..r-1] — (assumption)
(e2) RW≤ns⊖vr≤N-1 — (e1, s≥r (from A_1), c, B_1, Lemma 1)
(e3) Sink''=Sink, r''=r — (e2, Rec_D)

(e) e1 ⇒ $A_{0\text{-}3,8}''$ — (e1, e3, $A_{0\text{-}3,8}$)

(f1) uns in [r+1..s-1] — (assumption)
(f2) 0<ns⊖<RW **and** uns=r+ns⊖vr — (f1, s≤r+N (from A_1), c, B_1, Lemma 1)

(f3.1) Sink[uns]≠empty — (assumption)
(f3.2) Sink[uns]=Source[uns], uns≠r — (f1, f2, f3.1, A_2, B_0)
(f3.3) Sink''=Sink, r''=r — (f2, f3.2, Rec_D)

(f3) f3.1 ⇒ $A_{0\text{-}3,8}''$ — (f3.1, f3.3, $A_{0\text{-}3,8}$)

(f4.1) Sink[uns]=empty (assumption)
(f4.2) Sink[uns]"=data=Source[uns] (f1, f2, f4.1, Rec_D, c)
(f4.3) (for m≠n)[Sink[m]=Sink[m]"] **and** r"=r (f1, f2, f4.1, Rec_D)

(f4) f4.1 ⇒ $A_{0\text{-}3,8}$" (f4.1, f4.2, f4.3, $A_{0\text{-}3}$, A_8)

(f) f1 ⇒ $A_{0\text{-}3,8}$" (f1, f3, f4)

(g1) uns=r (assumption)
(g2) ns⊖vr=0 (g1, c, B_1)
(g3) Sink[r]=data=Source[r] (g1, g2, c, Rec_D)
(g4) r">r, Sink[r"]=empty, (g2, g3, SinkData)
(for all n in [r+1..r"-1])[Sink[n]"=Sink[n]≠empty],
(for all n in [r+RW..r"+RW-1])[Sink[n]"=empty]
(g5) r"≤s (g4, A_3)
(g6) A_0" (A_0, g4, g5, A_2)
(g7) $A_{1\text{-}3,8}$" (g4, g5, $A_{1\text{-}3,8}$)

(g) g1 ⇒ $A_{0\text{-}3,8}$" (g6, g7)
(h) $A_{0\text{-}3,8}$" (d, e, f, g)

AcceptData: $A_{0,2}$ not affected.

(a) s"=s+1, vs⊖va≤N-RW-1, a"=a (AcceptData)
(b) A_{10}" (a, A_{10})
(c) $A_{1,3}$" (a, $A_{1,3}$)
(d) A_8" (A_8, E_2)
(e) A_9" (A_9, E_1)

End of proof

Proof of invariance of B_6

Initial Conditions:

(a) B_6 ($\mathbf{z}_1$=null)

Channel errors, Rec_D:

(a) B_6" (B_6, $\langle m,t\rangle$ in $\mathbf{z}_1''$ ⇒ $\langle m,t\rangle$ in $\mathbf{z}_1$)

Send_D:

(a) i in [0..s-a-1], $\mathbf{z}_1''=(\mathbf{z}_1,\langle(D,data,ns,a+i),0\rangle)$,DTimeG[a+i]"=0, (Send_D)
(for all n≠a+i)[DTimeG[n]=DTimeG[n]"]
(b) B_6" (a, B_6)

Ideal time event:

(a) B_6" (B_6, age"=age+1, DTimeG"=DTimeG+1)

Other events do not affect B_6.

End of proof

Proof of Lemma 4.
Initial Conditions $\Rightarrow$ s=0 $\Rightarrow$ A_{11}.
The ideal time event, Send_D, and AcceptData are the only events that affect A_{11}.
Ideal time event:

(a) A_{11}" $\qquad$ (A_{11}, DTimeG"=DTimeG+1)

Send_D:

(a) A_{11}" $\qquad$ ($A_{1,10,11}$, Send_D)

AcceptData **and** E_4:

(a) A_{11}" $\qquad$ (s"=s+1 (AcceptData), A_{11}, E_4)
(b) (s≥N-RW **and** DTimeG[s-N+RW]≥MaxDelay$_1$) $\Rightarrow$ (D,data,ns,s-N+RW) not in $\mathbf{z}_1$ $\qquad$ (B_6, timer axiom of C_1)
(c) (s≥N-RW **and** DTimeG[s-N+RW]=Off) $\Rightarrow$ (D,data,ns,s-N+RW) not in $\mathbf{z}_1$ $\qquad$ (B_6)
(d) E_4 $\Rightarrow$ (D,data,ns,s-N+RW) not in $\mathbf{z}_1$ $\qquad$ (b, c)
(e) E_1 $\qquad$ (d, E_4, A_9)

End of proof

Proof of invariance of $B_{7\text{-}8}$.
Initial Conditions:

(a) $B_{7\text{-}8}$ $\qquad$ (r=0, $\mathbf{z}_1$=null)

Send_D, AcceptData, Local time event of P_1: $B_{7\text{-}8}$ not affected.

Channel errors, Rec_ACK: B_7 not affected.

(a) B_8" $\qquad$ (B_8, <m,t> in $\mathbf{z}_2''$ $\Rightarrow$ <m,t> in $\mathbf{z}_2$)

Ideal time event:

(a) $B_{7\text{-}8}$" $\qquad$ ($B_{7\text{-}8}$, age"=age+1, RTimeG"=RTimeG+1)

Send_ACK: B_7 not affected.

(a) $\mathbf{z}_2''$=($\mathbf{z}_2$,<(ACK,vr,r),0>) $\qquad$ (Send_ACK)
(b) B_8" $\qquad$ (a, $B_{7\text{-}8}$)

Rec_D:

(a) r"≥r **and** RTimeG[r]"=RTimeG[r+1]"= ... =RTimeG[r"-1]=0 $\qquad$ (Rec_D)
(b) $B_{7\text{-}8}$" $\qquad$ (a, $B_{7\text{-}8}$)

End of proof

Proof of invariance of B_9.
Initial Conditions:

(a) B_9 (a=0)

Rec_ACK:

(a) $a" \geq a$ **and** ATimeG[a]"= ... =ATimeG[a"-1]"=0 (Rec_ACK)
(b) B_9" (a, B_9)

Ideal time event:

(a) B_9" (B_9, ATimeG"=ATimeG+1)

Other events do not affect B_9
End of proof

Proof of Lemma 5.
Initial Conditions ⇒ s=a=0 ⇒ $A_{12\text{-}13}$.

The ideal time event, Rec_D, Rec_ACK, and AcceptData are the only events that affect $A_{12\text{-}13}$.

Ideal time event:

(a) $A_{12\text{-}13}$" ($A_{12\text{-}13}$, ATimeG"=ATimeG+1, RTimeG"=RTimeG+1)

Rec_D: A_{13} not affected.

(a) $r" \geq r$, RTimeG[0..r-1]"=RTimeG[0..r-1] (Rec_D)
(b) A_{12}" (a, A_{12}, $a \leq r$ (from A_1))

Rec_ACK:

(a) $a" \geq a$, ATimeG[0..a-1]"=ATimeG[0..a-1], ATimeG[a..a"-1]"=0 (Rec_ACK)
(b) $a \geq s\text{-}N+RW$ (A_{10})
(c) A_{13}" (a, b, A_{13})
(d) $a" \leq r$ (A_1")
(e) A_{12}" (a, d, A_{12}, B_7)

AcceptData **and** E_5: A_{12} not affected.

(a) A_{13}" (s"=s+1 (AcceptData), A_{13}, E_5)
(b) $s \leq a+N\text{-}RW\text{-}1 \leq a+N\text{-}2$ (AcceptData, A_{10})
(c) (for n in [0..s-N+1])[RTimeG[n]>$MaxDelay_2$] (b, $A_{12\text{-}13}$")
(d) E_2 (c, $B_{7\text{-}9}$, Timer axiom for C_2)

End of proof

Appendix B

Proof of invariance of $B_{12\text{-}13}$.

Initial Conditions:

(a) $B_{12\text{-}13}$ (s=0, DTimer[0..N-1]=Off)

Send_D, AcceptData and the time events are the only events that affect $B_{12\text{-}13}$.

Observe that [max(0, s-N+RW)..max(s-RW-1, N-1)] always has N consecutive integers; if s<N-RW then it equals [0..N-1] else it equals [s-N+RW..s-1+RW]. Thus, the following holds:

(i) (for all nonnegative integer j)(for all n,m in [max(0, j-N+RW)..max(j+RW-1, N-1)])
[n=m ⇔ (n mod N)=(m mod N)]

Also observe that the following holds (it holds initially, and is preserved by AcceptData and Send_D):

(ii) DTimeG[s..∞] = Off

Send_D:

(a) (for some i in [0..s-a-1])[DTimer[va⊕i]"=DTimeG[a+i]"=0 (Send_D)
and (for all k≠i)[DTimer[va⊕k]=DTimer[va⊕k]"
and DTimeG[a+k]=DTimeG[[a+k]"]

(b) a+i in [max(0, s-N+RW)..s-1] (A_{10}, i)

(c) (a+i) mod N = va⊕i (B_1)

(d) $B''_{12\text{-}13}$ ($B_{12\text{-}13}$, a, b, c)

AcceptData:

(a) vs"=vs⊕1, s"=s+1 (AcceptData)
DTimeG[0..s-1]"=DTimeG[0..s-1], DTimer"=DTimer

(b) DTimer[vs]=Off (vs=s mod N (from B_1), B_{13})

(c) DTimeG[s]=Off (from (ii))

(d) B''_{12} (a, b, c, vs=s mod N (from B_1), B_{12})

(e) DTimer[vs⊕RW]=Off (AcceptData)

(f) B''_{13} (a, e, vs=s mod N (from B_1), B_{13})

Ideal time event: B_{13} not affected.

(a) B''_{12} (B_{12}, DTimeG"=next(DTimeG), defn. of next)

Local time event for P_1:

(a) DTimer"=next(DTimer) (local time event for P_1)

(b) DTimer[i]=Off ⇒ DTimer[i]"=Off (defn of next)

(c) B_{13}" (B_{13}, b, a)

(d) 0≤DTimer[i]<$MDelay_1$ ⇒ DTimer[i]"=DTimer[i]+1 (defn of next)

(e) DTimer[i]=$MDelay_1$ ⇒ DTimer[i]=Off (defn of next)

(f) DTimer[n mod N]=$MDelay_1$ **and** n in [max(0, s-N+RW)..s-1] (e, B_{12},
    ⇒ DTimeG[n]>$MaxDelay_1$) (Started at property)

(g) B''_{12} (d, e, f, B_{12})

End of proof

Proof of invariance of B_{14} and B_{15}.

Initial Conditions:

(a) $B_{14\text{-}15}$ (s=a=0, ATimer[0..N-1]=Off)

AcceptData, Rec_ACK and the time events are the only events that affect B_{14} and B_{15}.

Observe that [max(0, s-N+1)..max(s, N-1)] always consists of N consecutive integers: if s<N-1 then the interval equals [0..N-1], else it equals [s-N+1..s]. Thus, the following holds:

(i) (for all nonnegative j)(for all n,m in [max(0, j-N+1)..max(j, N-1)])
    [n=m ⇔ (n mod N)=(m mod N)]

Also from A_{10}, we have

(ii) [max(0, s-N+1)..a-1] ⊂ [max(0, s-N+1)..max(s, N-1)]

AcceptData:

(a) vs" = vs⊕1, s" = s + 1 (AcceptData)
(b) B''_{14} (a, B_{14})
(c) ATimer[vs⊕1]=Off (AcceptData)
(d) ATimer[s" mod N]=Off (a, c, B_1)
(e) B''_{15} (d, B_{15})

Rec_ACK:

(a) a ≤ a" ≤ s ≤ a+N-RW (A_{10}, Rec_ACK)
(b) (for all n in [a..a"-1])[ATimer[n mod N]"=ATimeG[n]"=0] (Rec_ACK, B_1)
(c) B''_{14} (B_{14}, b, a, i, ii)
(d) B''_{15} (B_{15}, a, ii)

Ideal time event: B_{15} not affected.

(a) B''_{14} (B_{14}, ATimeG"=ATimeG+1)

Local time event of P_1:

(a) ATimer"=next(ATimer) (local time event of P_1)
(b) ATimer[i]=Off ⇒ ATimer[i]"=Off (defn of next)
(c) B''_{15} (B_{15}, a, b)
(d) 0≤ATimer[i]<$MDelay_2$ ⇒ ATimer[i]"=ATimer[i]+1 (defn of next)

(e) ATimer[i]=$MDelay_2$ ⇒ ATimer[i]"=Off (defn of next)

(f) ATimer[n mod N]=$MDelay_2$ **and** n in [max(0, s-N+1)..a-1] (B_7, Started at property)
⇒ ATimeG[n]>$MaxDelay_2$

(g) B''_{14} (B_{14}, d, e, f)

End of proof

Proof of invariance of B_{16}.

Initial Conditions:

(a) B_{16} (a=0)

Channel errors, Local time event of P_1, Send_ACK, Rec_D: B_{16} not affected.

AcceptData, Send_D:

(a) (for all n in [0..a-1])[DTimeG]n]=DTimeG[n]"] (A_1, AcceptData **or** Send_D)

(b) ATimeG=ATimeG" (AcceptData **or** Send_D)

(c) B_{16}" (a, b, B_{16})

Rec_ACK:

(a) (for all n in [a..a"-1])[ATimeG[n]"=0] (Rec_ACK)

(b) ATimeG[0..a-1]"=ATimeG[0..a-1] (Rec_ACK)

(c) (for all n in [a..a"-1])[DTimeG[n]≥0] (A_{12}")

(d) B_{16}" (a, b, c, B_{16})

Ideal time event:

(a) B_{16}" (B_{16}, DTimeG"=DTimeG+1, ATimeG"=ATimeG+1)

End of proof

Appendix C

Proof of invariance of D_2

We first observe that the following is invariant (proof is trivial):

(i) $\text{TnRecd}[a] \subseteq [0..\text{TestCount}]$ **and** $\text{TnRecd}[a+1..\infty] \subseteq \{0\}$ **and**
$((D,\text{data},\text{ns},\text{uns},\text{tn}) \text{ in } \mathbf{z}_1$ **and** $\text{tn} \neq 0) \Rightarrow (\text{uns} \leq a$ **and** $(\text{uns}=a \Rightarrow \text{tn in } [1..\text{TestCount}]))$

The proof of D_2's invariance now follows. The proof assumes that $A_{0\text{-}13}$ (see Table 1) is invariant; this holds because the real-time protocol system is a refinement of the earlier protocol systems.

Initial Conditions $\Rightarrow$ TestCount=0 $\Rightarrow$ D_2

Send_D, Local time events of P_1 and P_2, channel errors do not affect D_2.

AcceptData:

(a) $s>a \Rightarrow D_2''$ — (D_2 not affected (from AcceptData))
(b) $s=a \Rightarrow \text{TestCount}''=0 \Rightarrow D_2''$ — (AcceptData)
(c) D_2'' — (a, b, $s \geq a$ (from A_1))

Send_Test_D:

(a) $\text{TestTimerG}''=0$, $\text{TestCount}''=\text{TestCount}+1 \geq 1$ — (Send_Test_D)
(b) $\text{TestCount}=0 \Rightarrow \text{TnRecd}[a] \subseteq \{0\}$ — (from (i))
$\Rightarrow D_{2.1}'' \Rightarrow D_2''$ — (SCount$\geq$0, a)

(c1) $\text{TestCount} \geq 1$ — (assumption)
(c2) $\text{TestTimer} = \text{RTripDelay}$ — (Send_Test_D)
(c3) $\text{TestTimerG} > \text{RoundTripDelay}$ — (c1, c2, D_1, Started at property)
(c4) $\text{TestCount} \leq \text{SCount}_1[a] + \text{SC}_2[a+1]$ — (c1, c3, $D_{2.4}$)
(c5) $D_{2.1}''$ — (c1, c4, a, D_3)

(c) $\text{TestCount} \geq 1 \Rightarrow D_2''$ — (c1, c5)
(d) D_2'' — (b, c)

Rec_ACK:

(a) $s''=s \geq a'' \geq a$ — (Rec_ACK, A_1, A_1'')
(b) $a''=a \Rightarrow D_2''$ — (D_2 not affected (from Rec_ACK))
(c) $a''>a \Rightarrow \text{TestCount}''=0 \Rightarrow D_2''$ — (Rec_ACK)
(d) D_2'' — (b, c)

Rec_D:

(a) $\mathbf{z}_1 = (<D,\text{data},\text{ns},\text{uns},\text{tn}), \text{age}>, \mathbf{z}_1'')$ — (Rec_D)
(b) $D_{2.4} \Rightarrow D_{2.4}''$ — (Rec_D)

(c) $r'' \geq r$ (A_1, A_1'', Rec_D)

(d) $D_{2.2} \Rightarrow$ SendACKTimerG $\neq$ Off

$\Rightarrow$ SendACKTimerG'' = SendACKTimerG (Rec_D)

$\Rightarrow D_{2.2}''$ (c, Rec_D)

(e) $D_{2.3} \Rightarrow D_{2.3}''$ (c, Rec_D)

(f) $D_{2.1}$ **and** uns=a **and** tn=TestCount **and** SendACKTimer=Off $\Rightarrow D_{2.2}''$ (Rec_D)

(g) $D_{2.1}$ **and** uns=a **and** tn=TestCount **and** SendACKTimer$\neq$Off $\Rightarrow D_{2.2}''$ (Rec_D)

(h) $D_{2.1}$ **and** not (uns=a **and** tn=TestCount)) $\Rightarrow D_{2.1}''$ (Rec_D)

(i) D_2'' (b, e, f, g, h)

Send_ACK:

(a) SendACK=True, SendACKTimerG$\leq$MaxResponseTime, $\mathbf{z}_2'' = (\mathbf{z}_2, <(\text{ACK,vr,r}), 0>)$ (Send_ACK, timer axiom for SendACKTimer, started at property)

(b) ($D_{2.1} \Rightarrow D_{2.1}''$), ($D_{2.3} \Rightarrow D_{2.3}''$), ($D_{2.4} \Rightarrow D_{2.4}''$) (Send_ACK)

(c) $D_{2.2} \Rightarrow D_{2.3}''$ (a, Send_ACK)

Ideal time event:

(a) ($D_{2.1}$ **and** TestTimerG$\neq$Delay$_1$) $\Rightarrow D_{2.1}''$ (ideal time event)

(b) ($D_{2.1}$ **and** TestTimerG=Delay$_1$) $\Rightarrow D_{2.2}''$ (ideal time event)

(c) $D_{2.2} \Rightarrow 0 \leq$ SendACKTimerG $<$ MaxResponseTime (timer axiom

$\Rightarrow D_{2.2}''$ for SendACKTimer, started at property)

(d) ($D_{2.3}$ **and** TestTimerG$\neq$sendtime+Delay$_2$) $\Rightarrow D_{2.3}''$ (ideal time event)

(e) ($D_{2.3}$ **and** TestTimerG = sendtime+Delay$_2$) $\Rightarrow D_{2.4}''$ (ideal time event)

(f) $D_{2.4} \Rightarrow D_{2.4}''$

(g) D_2'' (a, b, c, d, e, f)

End of proof