A Periodic State Exchange Protocol and Its Verification

Mohamed G. Gouda, Arun N. Netravali, *Fellow, IEEE*, and Krishnan Sabnani, *Fellow, IEEE*

Abstract— **We present an elegant protocol for reliably transmitting data messages from a sender to a receiver over a high-speed network that may reorder, lose, or corrupt messages. The protocol is based on a new principle that calls for the periodic exchange of state information between the sender and receiver. (In traditional protocols, state information is exchanged only when significant events occur.) Our formal definition of the protocol is abstract and does not include explicit timing information such as the rate of sending state information. The abstract definition makes our formal verification of the protocol simple and based solely on well-established concepts: invariants, well-foundedness, and action fairness. We use the formal definition of the protocol and its proof of correctness to deduce the required timing information. In particular, we show that the rate of sending state information is at most $(m - 1)/2T$ where m is a measure of the memory size in the sender, and T is an upper bound on the required time for one message to be sent, propagated, and received between the sender and receiver.**

I. Introduction

A NOVEL principle for making transport protocols "lightweight" was developed recently by Netravali, Roome, and Sabnani. In traditional transport protocols, changes in the state are exchanged between the communicating entities when and only when a significant event, such as the delivery of an incorrect message, occurs. By way of contrast, the new principle calls for frequent and periodic exchange of state information between the communicating entities. This periodic exchange of state information reduces the number and complexity of recovery procedures to be invoked when state or control messages are lost or received incorrectly. We have previously described some protocols based on this principle in [1] and [2], discussed their implementation in [2], and reported their performance analysis in [3] and [4]. These results have already affected new proposals for protocol standards, see, for instance, [5].

Applying the principle of periodic state exchange causes a large number of (state) messages to be generated in the network. Therefore, this principle is better suited for a high-speed network that usually has untapped throughput. In effect, this principle tends to convert the untapped throughput of the network into utilized throughput and lowered response time.

Paper approved by S. Aggarwal, the Editor for Communication Software of the IEEE Communications Society. Manuscript received Apr. 16, 1992; revised August 11, 1993. M. G. Gouda was supported in part by a grant from the Texas Advanced Technology Program 1992–1994.

M. G. Gouda is with the Department of Computer Sciences, University of Texas at Austin, Austin, TX 78712 USA.

A. N. Netravali is with AT&T Bell Laboratories, Murray Hill, NJ 07974 USA.

K. Sabnani is with AT&T Bell Laboratories, Holmdel, NJ 07733 USA.

IEEE Log Number 9413163.

In this paper, we present an elegant protocol that uses the principle of periodic state exchange for reliably transmitting data messages from a sender to a receiver over a network that can reorder, lose, or corrupt transmitted messages. We also give a formal proof for the correctness of the protocol. Our objective of this exercise is two-fold. First, we want to provide more evidence of the effectiveness of periodic state exchange as a basis for designing lightweight protocols in high-speed networks. Second, we want to give a detailed and logical explanation, through the correctness proof, on how this principle works.

The basic idea of our protocol is simple. The receiver has a circular buffer where received data messages are stored until they are removed by the host of the receiver. Periodically, the receiver sends the current state of the circular buffer to the sender. The periodic state messages serve two purposes. First, they inform the sender of whether previously sent data messages have been received by the receiver and stored in the circular buffer, or have been lost and need to be resent. Second, the periodic state messages inform the sender of the available space in the circular buffer where the sender can send new data messages.

The apparent simplicity of this protocol is somewhat deceiving. A straightforward definition of the protocol would include a real-time clock in the receiver to regulate the rate at which periodic state messages are sent by the receiver. The protocol definition would, then, include estimates for the needed times, as measured on the real-time clock, to execute different actions in the protocol, such as sending or receiving a message, or updating the contents of the circular buffer. All these details would complicate the formal definition of the protocol, and, even worse, complicate its formal verification.

To avoid this complexity, we resort to abstraction in our formal definition of the protocol. Instead of introducing a real-time clock to the protocol definition, we use an abstract timeout action for sending the state of the circular buffer. This action is abstract because its execution is not triggered by the advance of real time (as it should be implemented), but by the occurrence of a certain global condition. The use of an abstract timeout action instead of a real-time clock makes both the protocol definition and its correctness proof tractable and simple. Later (in Section V), we discuss how to use a real-time clock to detect the occurrence of the global condition in the timeout action.

Notice that many protocols in high-speed networks are based on rate control and the utilization of real-time clocks, see for instance, [6] and [7]. In that respect, these protocols

resemble the protocol that we describe in this paper. Thus, the abstraction that we use in specifying the real-time clock of our protocol can be also used in specifying the real-time clocks of other protocols in high-speed networks.

The rest of the paper is organized as follows. A detailed, yet informal, description of the protocol is given in Section II. Then, a formal definition of the protocol is presented in Section III. In Section IV, we give a correctness proof establishing the safety, progress, and fault-tolerance properties of the protocol. Then in Section V, we discuss a number of issues related to implementation of the protocol from its formal definition. Concluding remarks are given in Section VI.

II. Description of the Protocol

The protocol that we consider in this paper consists of two processes called receiver and sender. The receiver process P has a circular buffer where it stores the data messages that it has received from the sender process Q until the messages are removed by the host of the receiver.

Periodically, the receiver P sends the current state of its circular buffer to the sender Q. On receiving a state message, Q can decide which data messages have been lost during transmission, and so need to be retransmitted. It can also decide how many new data messages it can send to P. The state of the circular buffer consists of three components:

$$(rcvd,\ pr,\ pt)$$

where $rcvd$ is a boolean map of the circular buffer, and pr and pt are two pointers that always point to different locations of the circular buffer. These two pointers divide the circular buffer into two nonempty regions: A "can–receive region" from pr to just before pt and a "cannot–receive region" from pt to just before pr. Q can send and P can receive data messages in the first region, but they cannot do so in the second region where old messages still reside waiting to be removed by the host of the receiver.

Because pr and pt always point to different locations of the circular buffer, at least a third location is needed in order that one of these pointers can be incremented (in a circular fashion) without becoming equal to the other pointer. Thus, the number n of locations in the circular buffer is at least three, i.e., $n \geq 3$.

Sent data messages can be reordered, i.e., received by the receiver in an order different from the one in which they were sent by the sender. To counter this problem, each sent message is given a sequence number in the range $0 \cdots n-1$ to indicate the location in the circular buffer where the message is to be stored after it is received. Hence, even if data messages are received out of order, they still end up in the circular buffer in the same order in which they were sent.

When the sender Q receives a state message that describes the state of the circular buffer, it can decide which data messages have been lost during earlier transmissions and so need to be retransmitted. The rule for making this decision is simple. Q keeps a variable count $[k]$ for each data (k) message it sends. This variable counts the number of state messages that Q has received after sending data (k), and from which Q has concluded that data (k) has not been received by the receiver. When count $[k]$ equals a certain value m, then Q concludes correctly that the sent data(k) message has been lost, and so resends it and resets count $[k]$ to 0.

Problems may arise if m or more state messages can be present in the channel from P to Q at the same instant. Consider the following scenario. Process Q sends a new data (k) message and makes count $[k] = 0$ at an instant when there are m state messages in the channel from P to Q. Shortly, Q receives all the state messages that were in the channel prior to sending data (k), and so detects in each of them that P has not yet received data (k). This causes count $[k]$ to reach the value m and Q to reach the wrong conclusion that the sent data (k) message has been lost and should be resent. To avoid such a scenario, the rate of sending state messages from P to Q should ensure that at any instant at most $m-1$ state messages can be in the channel from P to Q. Therefore, $m \geq 2$.

Problems may arise if state messages are reordered (i.e., received in an order different from the one in which they were sent). Consider the following scenario. The sender Q sends a data (k) message to the receiver which receives it correctly and acknowledges its reception in two successive state messages. Process Q receives the first state message and resumes sending new data messages without receiving the second state message. This second message can still be received later, after Q receives any number of future state messages (because message reorder is allowed). Because the sequence numbers of data messages are bounded, process Q eventually returns to sequence number k and sends a new data (k) message. Suppose that this new data (k) message is lost, but Q receives the old state message that acknowledges the reception of a data (k) message by the receiver. In this case, Q decides not to resend the lost data (k) message, causing a permanent loss. Even if state messages are assigned bounded sequence numbers, a similar scenario can still lead to a permanent message loss.

In order to counter the effects of state message reorder, state messages are tagged with unbounded, monotonically increasing sequence numbers. Process Q keeps track of the highest sequence number qk that it has received in a state message. When Q receives a state message, it compares its sequence number with qk. If the sequence number is higher than qk, the message is processed as usual. Otherwise, it is discarded.

So far, we have argued that some problems may arise during protocol execution if $n < 3$, $m < 2$, or more than $m-1$ state messages are in the channel from P to Q, or if data messages have no sequence numbers, or if state messages have bounded sequence numbers. This reasoning has necessitated for us to make the following design decisions.

1) $n \geq 3$.
2) $m \geq 2$.
3) At most $(m-1)$ state messages are ever in the channel from P to Q.
4) Data message have sequence numbers in the range $0 \cdots n-1$.
5) State messages have unbounded sequence numbers.

These design decisions only guarantee that the problems we have identified in our discussion will not arise during protocol

execution. They do not guarantee that other potential problems will not arise during the execution. To ensure that the protocol is problem-free, we have only one choice: formally verify that the protocol is correct. However, we cannot formally verify an informal description of the protocol like the one given above; only formal protocol definitions can be formally verified. Therefore, we formally define the protocol in Section III, then formally verify it in Section IV.

A. Notation

We adopt the following notation in our formal definition of the protocol in the next section.

$+_n$ and $-_n$	denote respectively addition and subtraction modulo n.
bet (x, y, z)	is a predicate over three variables x, y, and z whose values range over $0 \cdots n-1$. This predicate is true iff $(x \neq z) \wedge (y = x \vee y = x +_n 1 \vee y = x +_n 2 \vee \cdots \vee y = z -_n 1)$.

III. Formal Definition of the Protocol

As mentioned earlier, the protocol consists of two processes: process P for the receiver and process Q for the sender. Each process is defined by a set of local variables and a set of actions.

process <process name>
var <Pascal-like declarations of local variable>
begin
<action>
▯ <action>
...
▯ <action>
end

Each action is of the form:
<label> : <guard> → <statement>

The label is a number that uniquely identifies the action. The guard is either a boolean expression or a receive statement of the form: **rcv** g, for some message g. The statement is defined recursively as one of the following:

a **skip** statement,

an assignment statement,

a send statement of the form: **send** g,

a sequence of statements of the form:
⟨statement⟩; ⟨statement⟩,

a conditional statement of the form:
if ⟨condition⟩ **then** ⟨statement⟩ **fi**, or

an iterative statement of the form:
for ⟨condition⟩ **do** ⟨statement⟩ **od**

There are two one-directional channels between processes P and Q. At each instant during protocol execution, each channel has a sequence of messages. The message sequence in the channel from P to Q contains all state messages that have been sent by P but not yet received by Q. The message sequence in the channel from Q to P contains all data messages that have been sent by Q but not yet received by P. The sending of a message from P to Q (or from Q to P) consists of adding the message at the tail of the message sequence in the channel from P to Q (or from Q to P, respectively). The receiving of a message from P to Q (or from Q to P) consists of removing the message at the head of the message sequence in the channel from P to Q (or from Q to P, respectively).

The message sequence in a channel is updated by executing send or receive statements. It is also updated by the occurrence of the following types of faults.

1) *Message Reorder*:
   Two messages in the message sequence are interchanged.
2) *Message Loss*:
   One message disappears from the message sequence.
3) *Message Corruption*:
   One message in the message sequence is replaced by the special message error.

A protocol state is defined by a value for each local variable in process P or Q, a sequence of state or error messages in the channel from P to Q, and a sequence of data or error messages in the channel from Q to P.

An action in process P (or Q) is enabled for execution at a protocol state s if the guard of the action is a boolean expression that evaluates to true at s, or if the guard is a receive statement **rcv** g, and g is the head message in the message sequence in the channel from Q to P (or from P to Q, respectively) at state s.

Execution of an enabled action in process P (or Q) depends on whether the guard of the action is a boolean expression or a receive statement. If the guard is a boolean expression, the action is executed by executing the statement of the action. If the guard is a receive statement, the action is executed by first receiving the head message from the message sequence in the channel from Q to P (or from P to Q, respectively), then executing the statement of the action.

A protocol computation is a maximal sequence of the form:

State.0; action.0; state.1; action.1; state.2; $\cdots$

where each state.i is a protocol state, and each action.i is an action in P or Q that is enabled for execution at state.i. Moreover, executing action.i starting at state.i yields state.$(i+1)$. The maximality of the sequence means that the sequence is infinite, or it is finite but no action is enabled for execution at its last state.

Next, we present the receiver process in Section III-A, and the sender process in Section III-B. Then, we discuss the timeout action in the receiver in Section III-C.

A. Receiver

The receiver P has the following local variables.

var $rcvd$: **array** $[0 \cdots n-1]$ **of boolean** $\{^* n \geq 3^*\}$
$pr, pt, i : 0 \cdots n-1$
pk : **integer**

Array $rcvd$ is used to record the current state of the circular buffer in P. Variables pr and pt are two pointers of array $rcvd$. They can be interpreted as follows:

pr is the sequence number of the next data message to be received.

pt is the sequence number of the next data message that cannot be received. (There is an old data message in that position waiting to be removed by the host of P.)

Thus, pr and pt are never equal, and they divide the elements of array $rcvd$ into two nonempty regions: can-receive region and cannot-receive region.

The can-receive region consists of those elements of array $rcvd$ whose indices are in the set $\{pr, pr +_n 1, \cdots, pt -_n 1\}$. This region corresponds to the positions in the circular buffer where newly received data messages can be stored. The cannot-receive region consists of those elements of array $rcvd$ whose indexes are in the set $\{pt, pt +_n 1, \cdots, pr -_n 1\}$. This region corresponds to those positions in the circular buffer that are still occupied by old data messages, i.e., by data messages that have been received earlier but have not yet been removed by the host of process P. Thus the value of an element of array $rcvd$ is as follows:

$$rcvd[k] = \begin{cases} \text{true if} & \text{bet}(pr, k, pt) \text{ and data}(k) \text{ has been received} \\ \text{false if} & \text{bet}(pr, k, pt) \text{ and data}(k) \text{ has not been received, or bet}(pt, k, pr) \end{cases}$$

Variable pk stores the sequence number of the last state message sent by P.

Process P has four actions: one action to send a state message, one action to receive a data message, one action to increment pr, and one action to increment pt. The four actions are described in detail next.

In the first action, process P sends a state message to Q. Each state message contains the current state of the circular buffer as defined by array $rcvd$ and the two variables pr and pt. Each state message also contains a unique sequence number pk.

$$1\text{: timeout} \rightarrow pk := pk + 1;\ \textbf{send}\ st(rcvd, pr, pt, pk).$$

We delay our discussion of the timeout predicate to Section III-C.

In the second action, process P receives one data (i) message from Q and updates the array element $rcvd[i]$ to true.

$$2\text{: } \textbf{rcv}\ \text{data}\,(i) \rightarrow rcvd[i] := \text{true}.$$

In our earlier discussion, we have stated that $rcvd[i]$ implies bet (pr, i, pt). Thus, in order for action 2 to be consistent with that statement, we need to show that at each reachable state of the protocol, each data (i) message in the channel from Q to P satisfies bet (pr, i, pt). This is done in Section IV-A as part of discussing the protocol invariant.

In the third action, process P recognizes that it has already received data (pr) and that $pr +_n 1 \neq pt$; it thus makes $rcvd[pr]$ false then increments pr.

$$3\text{: } rcvd[pr] \wedge pr +_n 1 \neq pt \rightarrow rcvd[pr] := \text{false};\ pr := pr +_n 1.$$

Note that this action cannot make pr equal pt.

The last action of P "simulates" the action of its host of removing data $[pt]$ from the circular buffer by incrementing pt.

$$4\text{: } pt +_n 1 \neq pr \rightarrow pt := pt +_n 1.$$

Again this action cannot make pr equal pt.

Process P can now be defined as follows:

process P
var $rcvd$: **array** $[0 \cdots n-1]$ **of boolean** $\{^*n \geq 3^*\}$
pr, pt, i: $0 \cdots n-1$
pk: **integer**
begin
1: timeout $\rightarrow pk := pk + 1$; **send** $st(rcvd, pr, pt, pk)$
[] 2: **rcv** data $(i) \rightarrow rcvd[i] :=$ true
[] 3: $rcvd[pr] \wedge pr +_n 1 \neq pt \rightarrow rcvd[pr] :=$ false; $pr := pr +_n 1$
[] 4: $pt +_n 1 \neq pr \rightarrow pt := pt +_n 1$
end

B. Sender

The local variables of process Q are as follows:

var ackd: **array** $[0 \cdots n-1]$ **of boolean**
qr, qt, qs, j: $0 \cdots n-1$
qk: **integer**
count : **array** $[0 \cdots n-1]$ **of** $0 \cdots m-1$ $\{^*m \geq 2^*\}$
b : **array** $[0 \cdots n-1]$ **of boolean**
r, t : $0 \cdots n-1$
k : **integer**

The four variables b, r, t, and k are used to store the fields of the last received state message. (Recall that P sends each state message with four fields: $rcvd$, pr, pt, and pk.) If the received sequence number k is bigger than qk, the highest received sequence number, then the values of b, r, t, and k are stored in variables $ackd$, qr, qt, and qk, respectively. Thus, array $ackd$ is used to store the last received value of array $rcvd$. Similarly, variables qr and qt are used to store the last received values of variables pr and pt, respectively. It is convenient to interpret variables qr, qt, and qs as follows.

qr is the sequence number of the next data message to be acknowledged.

qt is the sequence number of the next data message that cannot be sent (because the receiver P currently has no space to store it in the circular buffer).

qs is the sequence number of the next data message to be sent.

Therefore, bet (qr, qs, qt) always holds.

Array count is used in counting, for each sequence number k, the number of state messages that have been received by

process Q after it has sent the last data (k) message and before this last data (k) is acknowledged. Whenever count $[k]$ reaches a specified value m, Q recognizes that the last sent data (k) is lost and resends it.

Process Q has two actions. In the first action, Q receives one state message. If the message sequence number is at most qk, then the message is recognized as an old message and discarded. Otherwise, the message contents are used to update the local variables $ackd$, qr, qt, and qk. Then, Q updates array count and resends every data (j) message where bet (qr, j, qs) and count $[j] = 0$ and $\sim ackd[j]$. (Recall that the value of count $[j]$ ranges from 0 to $m-1$; thus count $[j]$ reaching the value m can be detected when count $[j] = 0$.)

```
5: rcv st(b, r, t, k) → if k > qk
       then ackd, qr, qt, qk := b, r, t, k;
         for j = qr ··· qs −n 1
         do count[j] := count[j] +m 1;
           if count[j] = 0∧ ∼ ackd[j]
           then send data(j)
           fi
         od
       fi
```

In the second action, process Q recognizes that the sequence number qs can be used to send the next data message, and so it attaches qs to a new data message and sends the message to process P, then resets count $[qs]$ to 0 and increments qs.

```
6: qs +n 1 ≠ qt →  send data(qs);
                   count[qs] := 0; qs :=  qs  +n 1
```

Process Q is as follows.

```
process Q
var    ackd: array [0 ··· n − 1] of boolean
       qr, qt, qs, j: 0 ··· n − 1
       qk: integer
       count: array [0 ··· n − 1] of 0 ··· m − 1
       b: array [0 ··· n − 1]
       r, t: 0 ··· n − 1
       k: integer
begin

5: rcv st(b, r, t, k)→ if k > qk
       then ackd, qr, qt, qk := b, r, t, k;
         for j = qr ··· qs − n1
         do count[j]  := count[j] +m 1;
           if count[j] = 0 ∧  ãckd[j]
           then send data(j)
           fi
         od
       fi
  6: qs +n 1 ≠ qt  →  send data(qs);
                      count[qs] := 0; qs := qs +n 1
end
```

C. Timeout Condition

As mentioned earlier, the receiver P executes its timeout action periodically. Therefore, the guard of this timeout action can be expressed by specifying the rate at which the action is to be executed. The problem of this direct approach is that it requires the introduction of the new concepts of "real time" and "clocks" to both the protocol definition and its subsequent verification. Thus, this approach tends to complicate both the definition and verification of the protocol.

Instead of this direct approach, we adopt an indirect approach for specifying the guard of the timeout action. The indirect approach consists of two steps. First, the guard of the timeout action is specified as a predicate that identifies the protocol states at which the timeout action can be executed. Second, this predicate is used to deduce the rate at which the timeout can be executed. The first step of this approach is carried out in the current section, while the second step is carried out in Section V-A.

Each execution of a timeout message causes one state message to be sent from P to Q. The timeout period between two successive sendings of state messages is chosen large enough such that the following two conditions hold.

1) The number of state messages in the channel from P to Q never exceeds $m-1$.
2) If process P sends a state message which when received by Q causes a data (k) message to be resent by Q, then the last data (k) message that Q has sent is no longer in the channel from Q to P (i.e., has been lost) when the state message is sent.

In order to state these two conditions formally, we adopt the following notation.

$\#C_{PQ}$	denotes the number of (state) messages in the channel from P to Q.
data $(k)\#C_{QP}$	denotes the number of data (k) messages in the channel from Q to P.

To guarantee condition 1), the timeout action should have the following guard:

$$(\#C_{PQ} < m-1).$$

Because the timeout action sends one state message from P to Q, this guard ensures that sending this message does not make the number of state messages in the channel from P to Q exceed $m-1$ (as required by condition 1).

To guarantee condition 2), the timeout action should have the following guard:

$$\begin{aligned}&(\text{For every } k \text{ in the range } 0 \cdots n-1:\\ &\quad (\sim rcvd[k] \wedge \text{count}\,[k] + \#C_{PQ} = m-1)\\ &\qquad \Rightarrow \text{data}\,(k)\#C_{QP} = 0\\ &)\end{aligned}$$

The antecedent of this condition implies that a data (k) message may be resent by Q when Q receives all state messages in the channel from P and Q plus one more. The consequent of this condition guarantees that if this happens and a data (k) message is resent by Q, no multiple copies of data (k) will be present in the channel from Q to P (as required by condition 2).

It follows that the guard of the timeout action is the conjunction of the above two guards. The timeout action can

then be written as follows:

1: **timeout**
$(\#C_{PQ} < m-1) \wedge$
(For every k in the range $0 \cdots n-1$:
$(\sim rcvd[k] \wedge \text{count}[k] + \#C_{PQ} = m-1)$
$\Rightarrow \text{data}(k)\#C_{QP} = 0$
$) \rightarrow pk := pk+1;$ **send** $st(rcvd, pr, pt, pk)$.

The reserved word **timeout** in the guard is syntactic sugar intended to highlight the fact that the action is a timeout action.

IV. Protocol Verification

Correctness of any protocol can be established by showing that the protocol satisfies three logical properties: safety, progress, and fault-tolerance. These three properties can be expressed in terms of some predicate C that assigns a value, true or false, to every state of this protocol.

1) *Safety*:
This property states that if the protocol is at any state where C holds, then executing any action in P or Q yields a state where C holds; i.e., C is closed under protocol execution.
2) *Progress*:
This property states that for every protocol computation that starts with a state where C holds, each of the three pointers pr, qs, and pt is incremented by one infinitely often along the computation.
3) *Fault-Tolerance*:
This property states that if the protocol is at any state where C holds, then each occurrence of message reorder, loss, or corruption in one of the two channels between P and Q yields a state where C holds, i.e., C is closed under fault occurrence.

A predicate C that satisfies these three properties defines a closed domain for protocol execution. (This domain is closed under both protocol execution and fault occurrence.) In this closed domain, safety and fault-tolerance are always guaranteed, and progress is guaranteed infinitely often. Such a predicate is referred to as a protocol closure [8], or a protocol invariant.

Note that our notion of fault-tolerance is based on the assumption that transmission faults (i.e., message reorder, loss, and corruption) do not occur continuously during any execution. In particular, the elapsed time during any protocol execution can be partitioned into successive fault and no-fault periods as follows:

Fault period; no-fault period; fault period; no-fault period; $\cdots$

During a fault period, any number of faults can occur, whereas during a no-fault period, no fault can occur. A fault period can be of any length, whereas a no-fault period should be long enough such that some progress can be made (i.e., at least one of the counters pr, qs, and pt can be incremented) during that period. Under these reasonable assumptions of fault occurrence, progress is guaranteed infinitely often despite the fact that faults can occur.

For our protocol, we adopt the following predicate as a candidate for a protocol closure:

$$C = c1 \wedge c2 \wedge c3 \wedge c4 \wedge c5 \wedge c6$$

where

$c1 =$ $\text{bet}(qr, qs, qt)$
$\wedge \text{bet}(pr, qs, pt)$
$\wedge \text{bet}(qr, pr, qs +_n 1)$
$\wedge \text{bet}(qs +_n 1, qt, pt +_n 1)$
$\wedge pk \geq qk$

$c2 =$ (For every k in the range $0 \cdots n-1$:
$rcvd[k] \Rightarrow \text{bet}(pr, k, qs)$)

$c3 =$ (For every k in the range $0 \cdots n-1$:
$\text{data}(k)\#C_{QP} \leq 1$)

$c4 =$ (For every data k message in the channel from
$\cdot Q$ to P:
$\sim rcvd[k] \wedge \text{bet}(pr, k, qs)$
)

$c5 = (\#C_{PQ} \leq m-1)$

$c6 =$ (For every $st(b, r, t, k)$ message in the channel from
$\cdot P$ to Q where $k > qk$:
$\text{bet}(r, qs, t)$
$\wedge \text{bet}(r, pr, qs +_n 1)$
$\wedge \text{bet}(qs +_n 1, t, pt +_n 1)$
$\wedge pk \geq k$
$\wedge$ (For every u in the range $0 \cdots n-1$:
$(\sim b[u] \wedge \text{bet}(r, u, t) \wedge$
$\text{count}[u] + \{st(b', r', t', k') | k \geq k' > kq\}\#C_{PQ}$
$= m$

$) \Rightarrow (\sim rcvd[u] \wedge \text{data}(u)\#C_{QP} = 0)$
)
)

The notation $\{st(b', r', t', k') | k \geq k'\}\#C_{PQ}$ denotes the number of $st(b', r', t', k')$ messages in the channel from process P to process Q whose sequence number k' is at most k.

In order to show that this C is indeed a protocol closure, we need to show that it satisfies the above safety, progress, and fault-tolerance properties. This is done in the next three sections, but first consider a protocol state that satisfies $d1 \wedge d2 \wedge d3 \wedge d4$ where

$d1 = (pr = qr = qs = 0 \wedge pt = qt = 1 \wedge pk = qk = 0)$
$d2 =$ (For every k in the range $0 \cdots n-1$: $\sim rcvd[k]$
$d3 = (\#C_{QP} = 0)$
$d4 = (\#C_{PQ} = 0)$.

Because $d1$ satisfies $c1$, $d2$ satisfies $c2$, $d3$ satisfies $c3 \wedge c4$, and $d4$ satisfies $c5 \wedge c5$, any protocol state that satisfies

$d1 \wedge d2 \wedge d3 \wedge d4$ also satisfies C. Therefore, the protocol can start executing at any state that satisfies $d1 \wedge d2 \wedge d3 \wedge d4$.

A. Proving Safety

To prove safety, it is sufficient to prove that if every action in P or Q is executed at a state where C holds, then C also holds at the resulting state. We give here the proof for only one action, action 2 in process P. (The proofs for all other actions are similar.)

Recall that action 2 is as follows:

$$2: \ \mathbf{rcv}\ \text{data}\,(i) \rightarrow rcvd[i] := \text{true}.$$

Because this action does not update any of the variables pr, pt, pk, qr, qs, qt, and qk, it cannot invalidate $c1$. Also this action does not add any data messages to the channel from Q to P, and so it cannot invalidate $c3$. Similarly, this action does not add any state message to the channel from P to Q, so it cannot invalidate $c5$. It remains to show that action 2 cannot invalidate $c2$, $c4$, and $c6$.

In order that $c2$ hold after executing action 2 (given that it holds before the execution), the condition $\text{bet}\,(pr, i, qs)$ should hold before the execution. But this condition holds before the execution because $c4$ holds before the execution.

In order that $c4$ hold after executing action 2 (given that it holds before the execution), the channel from Q and P should have exactly one $\text{data}\,(i)$ message before the execution. That this condition holds before the execution follows from the fact that $c3$ holds before the execution.

In order that $c6$ hold after executing action 2 (given that it holds before the execution), the condition

(For every $st(b, r, t, k)$ message in the channel from P to Q:

$$\sim (\sim b[i] \wedge \text{count}\,[i] + \#C_{PQ} = m)$$

)

should hold after the execution. But this condition holds before the execution because both $c6$ and $\text{data}\,(i)\#C_{QP} \neq 0$ hold before the execution. Moreover, this condition cannot be invalidated by the execution (of action 2). Therefore, this condition holds after the execution. This completes our proof that if action 2 is executed at a state where C holds, then C also holds at the resulting state.

B. Proving Progress

We need to prove that each of the variables pr, qs, and pt is incremented by one infinitely often along any protocol computation that starts with a state where C holds. First, we show that every protocol computation is infinite. Second, we show that along every infinite computation that starts with a state where C holds, each of the variables pr, qs, and pt is incremented by one infinitely often.

To prove the first part, note that at least one of two actions, action 1 in process P or action 5 in process Q, is enabled for execution at each protocol state. Therefore, each protocol computation is infinite.

To prove the second part, let c be an arbitrary infinite computation of the protocol, and assume that c starts with a state where C holds. From the safety property, C holds at every state in computation c. Therefore, the relation $\text{bet}\,(pr, qs, pt)$ holds at every state in c. Hence, it is enough to show that at least one of the variables pr, qs, or pt is incremented by one infinitely often along c. Because pointer pr is incremented by one in action 3, pointer pt is incremented by one in action 4, and pointer qs is incremented by one in action 6, we need to show that at least one of the actions 3, 4, or 6 is executed infinitely often along computation c.

We adopt the following fairness assumption.

Action Fairness: If any action is continuously enabled along some computation, then this action is eventually executed along that computation.

Along computation c, if any of the actions 3, 4, or 6 is enabled for execution, then this action will continue to be enabled for execution until it is executed by the action fairness assumption. Therefore, our proof obligation is reduced to showing that if computation c has a state where none of the actions 3, 4, and 6 is enabled for execution, then c has a subsequent state where one of these actions is enabled for execution.

Let S be a state in computation c where none of the action 3, 4, and 6 is enabled for execution. The following condition holds at state S.

$$(\sim rcvd[pr] \vee pr +_n 1 = pt) \wedge pt +_n 1 = pr \wedge qs +_n 1 = qt.$$

Because n is at least 3, which implies $\sim (pr +_n 1 = pt \wedge pt +_n 1 = pr)$, this condition is equivalent to

$$\sim rcvd[pr] \wedge pr +_n 1 \neq pt \wedge pt +_n 1 = pr \wedge qs +_n 1 = qt.$$

There are two cases to consider.

Case 1: There is a message data (pr) in the channel from Q to P at state S. In this case, action 2 is enabled for execution, and by the fairness condition is eventually executed. If the received message in action 2 is $\text{data}\,(pr)$, then $rcvd[pr]$ becomes true and actions 3 becomes enabled for execution. On the other hand, if the received message in action 2 is not $\text{data}\,(pr)$, then message $\text{data}\,(pr)$ is still in the channel from Q to P, but it is one message closer to the head message in the channel. Because action 2 is still enabled for execution, the cycle repeats. By well-foundedness, the $\text{data}\,(pr)$ message is eventually received and action 3 becomes enabled for execution.

Case 2: There is no message data(pr) in the channel from Q to P at state S. Starting from state S, actions 1 and 5 are executed infinitely often along c. The message $st(b, r, t, k)$ that are sent in executing action 1 satisfy the condition

$$\sim b[pr] \wedge r = pr \wedge t = pt \wedge k > qk \wedge \text{bet}\,(r, qs, t).$$

Receiving each of these messages in executing action 5 causes $\text{count}\,[pr]$ to be incremented by one modulo m. Eventually, $\text{count}\,[pr]$ becomes 0 and a message $\text{data}\,(pr)$ is sent. The resulting state satisfies Case 1) above.

C. Proving Fault-Tolerance

The protocol tolerates message reorder because every occurrence of message reorder keeps C unchanged.

The protocol also tolerates message loss because every occurrence of message loss does not invalidate C. In particular, a message loss does not invalidate $c1$ and $c2$ because neither of them depends on the contents of the two channels. A message loss does not invalidate $c4$ and $c6$ because neither of them depends on the number of messages in the two channels. Finally, a message loss does not invalidate $c3$ and $c5$ because each of them merely establishes an upper bound on the number of messages in a channel.

In order to make the protocol tolerate message corruption where a message in a channel may be corrupted into the special message error before being received, we need to add the following action to each of the two processes in the protocol.

$$\textbf{rcv error} \rightarrow \textbf{skip}.$$

Thus, each occurrence of message corruption is transformed into an occurrence of message loss which can then be tolerated by the protocol.

V. Protocol Implementation

In defining and verifying our protocol, we have made a number of design decisions that need to be realized in any implementation of the protocol. Three of these design decisions need special attention in their realization: the timeout action, the unbounded sequence numbers of state messages, and action fairness. In this section, we discuss how to implement each of these design decisions.

A. Implementing the Timeout Action

Recall that the timeout action is as follows:

$$\begin{aligned}
1: \ & \textbf{timeout} \\
& (\#C_{PQ} < m-1) \wedge \\
& (\text{For every } k \text{ in the range } 0 \cdots n-1: \\
& \quad (\sim rcvd[k] \wedge \text{count}\,[k] + \#C_{PQ} = m-1) \\
& \qquad \Rightarrow \text{data}\,(k) \#C_{QP} = 0 \\
&) \rightarrow pk := pk+1;\ \textbf{send}\ st(rcvd,\ pr,\ pt,\ pk).
\end{aligned}$$

This action can be implemented using a real-time clock in the receiver P. Instead of executing the statement of the action when the guard is true, the statement of the action is executed periodically, with a constant rate of r times per second as measured on the real-time clock in P. The problem now is to find an upper bound for r such that the guard of the timeout action is guaranteed to hold whenever the timeout action is executed based on the real-time clock.

The guard of the timeout action consists of two conjuncts; thus, we derive two upper bounds for r. The first bound guarantees that the first conjunct holds whenever the timeout action is executed. The second bound guarantees that the second conjunct holds whenever the timeout action is executed. We adopt the smallest of these two upper bounds because it guarantees that both conjuncts hold whenever the timeout action is executed.

Let T_1 denote an upper bound on the time for a state message to be sent by the receiver, propagated from the receiver to the sender, then received by the sender. Note that the existence of T_1 is based on the strong assumption that the time-to-live of a state message in the network is at most T_1. In other words, the network is assumed to discard a state message if this message is not received by the sender within T_1 time from the instant it is sent. Thus, the number of state messages that can exist simultaneously in the channel from P to Q is less than r^*T_1; i.e., $\#C_{PQ} < r^*T_1$. To ensure that the first conjunct $\#C_{PQ} < m-1$ holds whenever the timeout action is executed, it is sufficient to require that $(r^*T_1 \leq m-1)$. This condition can be written as

$$r \leq (m-1)/T_1. \tag{1}$$

Let T_2 denote an upper bound on the time for a data message to be sent by the sender, propagated from the sender to the receiver, then received by the receiver. (Note that the existence of T_2 is based on the strong assumption that the time-to-live of a data message in the network is at most T_2). Assume that a data (k) message is sent by the sender at time t. Then, the condition $(\sim rcvd[k] \wedge \text{count}\,[k] = 0 \wedge \#C_{PQ} < r^*d_1)$ holds at t. This data (k) message can remain in the channel from Q to P during the time period $[t,\ t+T_2)$. Because the timeout action can be executed at any instant during the period $[t,\ t+T_2)$, the guard of the timeout action should hold at every instant in that period. But at every instant during that period, both $\sim rcvd[k]$ and $\text{data}\,(k)\#C_{QP} = 1$ hold. Thus, the condition $(\text{count}\,[k] + \#C_{PQ} < m-1)$ should hold at every instant in the period. But the following condition holds at every instant in the period $(\text{count}\,[k] + \#C_{PQ} < r^*T_1 + r^*T_2)$. Therefore, it is sufficient to require that $(r^*T_1 + r^*T_2 \leq m-1)$. This condition can be written as

$$r \leq (m-1)/(T_1+T_2). \tag{2}$$

Condition (2) implies condition (1) because T_2 is positive. Thus, under the assumption that $T_1 = T_2 = T$, both (1) and (2) can be replaced by the following condition:

$$r \leq (m-1)/2T. \tag{3}$$

Note that m is a measure of memory size in the sender P. In particular, the number of bits needed to implement array count in P is $n^* \log m$. If m is made smaller in order to reduce the needed memory in P, then rate r of sending state messages is reduced, according to (3), and the responsiveness of the protocol to transmission failures is reduced.

B. Implementing Unbounded Sequence Numbers

The sequence numbers of state messages can be implemented using a large number of bits, say 32 b. When variable pk in the receiver has a large value, say 2^{31}, the receiver stops executing and requests a protocol reset. The effects of a protocol reset are many. All pointers to the circular buffer, pr, pt, qr, qs, and qt are returned to their initial values. The sequence numbers pk and qk are returned to 0. The message

buffers in the receiver and sender are emptied, and the two channels between the receiver and sender are also emptied. Then, protocol execution is resumed. Although the cost of a protocol reset is substantial, the fact that resets are needed only rarely during protocol execution makes the average cost of protocol execution acceptable.

C. Implementing Action Fairness

The action fairness assumption states that if any action is continuously enabled, then this action is eventually executed. One way to implement this assumption is to consider the actions of each process, one by one, in a fixed circular order. If a considered action is found to be enabled, then this action is executed and the next action in the circular order is considered. If a considered action is found to be not enabled, then this action is not executed, and the next action in the circular order is considered. Therefore, each action that is continuously enabled will be reached in order and will be found enabled and be executed, fulfilling the action fairness assumption.

VI. Concluding Remarks

Our main objective in this paper is to formally present and explain an elegant protocol based on the principle of periodic state exchange. The simplicity and effectiveness of our protocol should illustrate the viability of this principle as a basis for designing light protocols for high-speed networks.

An important lesson that has been demonstrated in this paper is the power of abstraction. We have based our formal definition of the protocol on an abstract timeout action rather than on a real-time clock. The result of this abstraction has been to keep both the formal definition of the protocol and its correctness proof simple. The introduction of a real-time clock came later in our discussion of the implementation of the timeout action. But by then we were armed by the correctness proof of the protocol; thus, we were able to easily deduce the required rate for sending state information. As mentioned in the introduction, the same abstraction can be applied to simplify the definition and verification of other real-time protocols [6] and [7] in high-speed networks. Similar experiences of the power of abstraction have been reported in [8] and [9].

The presentation of the protocol in this paper consists of four components. First, an informal description in Section II highlighted the main features of the protocol. Second, a formal definition in Section III gave a complete and formal definition of every feature of the protocol. Third, a correctness proof in Section IV provided detailed explanation and deep understanding of the protocol's execution. Fourth, implementation details in Section V demonstrated that the main abstractions in the formal definition can be realized with reasonable accuracy and cost. We believe that all protocols should be presented using these four components.

An important question concerning our protocol is whether the protocol is stabilizing [10]. It is shown in this paper that the protocol performs correctly if it computation starts at a state where C holds. To establish that the protocol is stabilizing, one needs to show that every computation of the protocol has a state where C holds. To establish that the protocol is not stabilizing, one needs to exhibit a (infinite) computation of the protocol where C never holds. A stabilizing protocol can tolerate failures or crashes that may place the protocol in arbitrary states. Although we strongly feel that the protocol is stabilizing, our efforts to prove (or disprove) this fact have so far failed.

Acknowledgment

The authors would like to thank E. Biersack, D. Kristol, and M. Merritt for reading earlier drafts of this paper and making a number of suggestions for improving the presentation. We are also thankful to the referees for their careful reading of the paper and suggesting additional improvements.

References

[1] K. Sabnani and A. N. Netravali, "A high-speed transport protocol for datagram/virtual circuit networks," in *Proc. ACM SIGCOMM '89 Symp.*, Austin, TX, Sept. 19–22, 1989, pp. 146–157.

[2] A. N. Netravali, W. R. Roome, and K. Sabnani, "Design and implementation of a high-speed transport protocol," *IEEE Trans. Commun.*, Nov. 1990.

[3] B. Doshi, P. Johri, A. N. Netravali, and K. Sabnani, "Performance of error and flow control for high-speed transport protocol," in *Proc. 13th Int. Teletraffic Congr.*, 1991.

[4] ——, "Error and flow control for a high-speed transport protocol," *IEEE Trans. Commun.*, vol. 40, 1992.

[5] S. Dravida, J. S. Swenson, and R. Zoccolillo, "A proposed convergence protocol to serve class C and class D users," *Contrib. T1S1*, Doc. T1S1.5/91-096, 1991.

[6] D. Clark, M. Lambert, and L. Zhang, "A high throughput bulk data transfer protocol," in *Proc. SIGCOMM'87 Symp.*, 1987.

[7] L. Zhang, "Virtual clock: A new traffic control algorithm for packet switched networks," *ACM Trans. Comput. Syst.*, vol. 9, no. 2, pp. 101–124, May 1991.

[8] M. G. Gouda, "Protocol verification made simple," *Comput. Networks and ISDN*, vol. 25, pp. 969–980, 1993.

[9] G. Brown, M. G. Gouda, and R. E. Miller, "Block acknowledgment: Redesigning the window protocol," *IEEE Trans. Commun.*, vol. 39, pp. 524–432, Apr. 1991.

[10] M. G. Gouda and N. Multari, "Stabilizing communication protocols," *IEEE Trans. Comput.*, vol. 40, pp. 448–458, Apr. 1991.

Mohamed G. Gouda was born in Cairo, Egypt. He received the B.S. degrees in engineering and mathematics, from Cairo University. He received the M.A. degree in mathematics from York University, and the Master and Ph.D. degrees in computing science from the University of Waterloo, Waterloo, Ont., Canada.

He moved to the U.S. where he worked for the Honeywell Corporate Technology Center for three years. In 1980, he moved to the University of Texas at Austin, and has settled there ever since. He currently holds The Mike A. Myers Centennial Professorship in Computing Science at the University of Texas at Austin. He has spent one summer at Bell Laboratories, Murray Hill, NJ, one summer at MCC, Austin, TX, and one winter at the Eindhoven Technical University in the Netherlands. His area of research is distributed and concurrent computing. In this area, he has been working on abstraction, nondeterminism, atomicity, convergence, stability, formality, correctness, efficiency, scientific elegance and technical beauty (not necessarily in that order).

Dr. Gouda was the founding Editor-in-Chief of the journal *Distributed Computing* (New York: Springer-Verlag, 1985). He was the Program Committee Chairman of the 1989 SIGCOMM Symposium sponsored by ACM. He was the first Program Committee Chairman for the International Conference on Network Protocols established by IEEE in 1993. He is an original member of the Austin Tuesday Afternoon Club.

Arun N. Netravali (S'67–SM'78–F'85) received the B.Tech. (Honors) degree from the Indian Institute of Technology, Bombay, India, in 1967, and the M.S. and Ph.D. degrees from Rice University, Houston, TX, in 1969 and 1970, respectively, all in electrical engineering. In 1994, he received an honrary doctorate from the Ecole Polytechnique Federale, Lausanne, Switzerland.

He is Vice President, Quality, Engineering, Software and Technologies (QUEST), and Communications Sciences Research Vice President at AT&T Bell Laboratories. He joined Bell Laboratories in 1972 as a Member of Technical Staff and became Head of the Visual Communications Research Department in 1978, Director of Computing Systems Research in 1983, and Communications Sciences Research Vice President in 1992 with added responsibility as a project manager for HDTV in 1990. He became Vice President of QUEST in 1994. He was at NASA from 1970 to 1972, where he worked on problems related to filtering, guidance, and control for the space shuttle. He has been an adjunct professor at the Massachusetts Institute of Technology since 1984, and has taught graduate courses at City College, Columbia University, M.I.T., and Rutgers University. He has served on the Digital Television Committees of the IEEE and the Society of Motion Picture and Television Engineers. He is an advisor to the Center for Telecommunications Research of Columbia University, the Swiss Federal Institute of Technology, Lausanne, Switzerland, and the Beckman Institute of the University of Illinois. He is the author of more than 120 papers and holds over 60 patents in the areas of computer networks, human interfaces to machines, picture processing, and digital television. He is the co-author of two books: *Digital Picture Representation and Compression* (New York: Plenum, 1987) and *Visual Communication Systems*, (New York: IEEE Press, 1989).

Dr. Netravali is a member of Tau Beta Pi and Sigma Xi and has served on the Editorial Board of the PROCEEDINGS OF THE IEEE from 1980 to 1984, and is currently an editor of several journals. He has organized and chaired sessions at several technical conferences, and was the Program Chairman of the 1981 Picture Coding Symposium and the 1990 International Conference on Pattern Recognition. He has edited several special issues for the IEEE, including two for the PROCEEDINGS on Digital Encoding of Graphics, and Visual Communication Systems, and one for the TRANSACTIONS ON PICTURE COMMUNICATION SYSTEMS. He is a Fellow of the AAAS and a member of the United States National of Academy of Engineering. He received the Donald G. Fink Award for the best review paper published in the PROCEEDINGS OF THE IEEE in 1980, the journal award for the best paper from the Society of Motion Pictures and Television Engineers in 1982, the L. G. Abraham Award for the best paper by the IEEE Communications Society in 1985 and in 1991, the Alexander Graham Bell Medal in 1991, the OCA National Corporate Employee Achievement Award in 1991, and Engineer of the Year Award from the Associates of Engineers from India in 1992. He serves on the New Jersey Governor's Committee on "Schools" programs.

Krishan Sabnani (M'84–SM'88–F'91) received the B.Tech. degree from Indian Institute of Technology, New Delhi, India, and the Ph.D. degree from Columbia University, New York, NY. In 1981, he joined AT&T Bell Laboratories after graduating from Columbia University.

He is currently a research department head at AT&T Bell Laboratories. His major area of interest is communication protocols and wireless networks.

Dr. Sabnani received the Leonard G. Abraham award from the IEEE Communications Society in 1991 and the Bell Laboratories Distinguished Technical Staff Award in 1990. He also received the President of India's Gold Medal and the Institute of Engineers (India) Gold Metal, both in 1975. He was a Cochairman of the Eighth International Symposium on Protocol Specification, Testing, and Verification held in Atlantic City, NJ, held in June 1988. He has also served as an Editor of the IEEE TRANSACTIONS ON COMMUNICATIONS and the IEEE TRANSACTIONS ON COMPUTERS. He has also served as a Guest Editor for the IEEE JOURNAL ON SELECTED AREAS IN COMMUNICATIONS (JSAC) and the *Computer Networks Journal*. He also serves on the editorial boards of IEEE/ACM TRANSACTIONS ON NETWORKING, *Journal of Systems Integration, Wireless Networks*, and *Journal of High Speed Networks*. He has served on the program committees of several conferences.