DELIVERY AND DISCRIMINATION: THE SEINE PROTOCOL[1]

M. G. Gouda[2], N. F. Maxemchuk[3],
U. Mukherji[3], and K. Sabnani[3]

Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712-1188

TR-88-41 November 1988
Revision: December 1989

Abstract

We present a family of protocols for data transmission from multiple identical senders to a single receiver. At each instant, every sender sends one bit, and the sent bits are or-ed together into one bit before being received by the receiver. If a sender has a data message to send, it sends the message bits one by one; otherwise it sends zero bits. Clearly, if the sending of two messages by two senders overlap in time, then the resulting "collision" can cause the receiver to receive a corrupted message, i.e., one that was not sent by either sender. The function of the protocol is to allow the receiver to detect and deliver those and only those messages that are not corrupted by collision. (In other words, the receiver acts as a discriminating seine that catches and delivers only uncorrupted messages; hence the title.) The seine protocol has been implemented in a class of fiber-optic local-area networks.

Keywords: balanced coding, communication protocol, collision, local area network, protocol verification.

---

[1]An early version of this paper has been presented at the ACM SIGCOMM 1988 conference.

[2]Work was supported in part by Office of Naval Research Contract N00014-86-K-0763, and in part by a contract with AT&T Bell Labs.

[3]AT&T Bell Laboratories, Murray Hill, NJ.

1. Introduction

In this paper, we present a family of protocols that can be used for data transmission in some fiber-optic networks, for example those described in [4] and [5]. In these networks, the bit streams from multiple senders are or-ed together on a bit-wise basis, into one stream. This stream is then delivered to every receiver in the network. The function of the protocol is to allow each receiver to recognize and extract from its incoming bit stream those and only those data messages that are not corrupted by collision with concurrently sent messages.

The protocols we discuss in this paper provide a delicate balance between delivery and discrimination. On one hand, they allow all those messages that are not corrupted by collision to be accepted and delivered. On the other hand, they are discriminate enough not to accept any message that is corrupted by collision.

The rest of this paper is organized as follows. In Section 2, we formally define the problem. Then in Section 3, we describe the rationale that we have applied to solve the problem. Our first solution, called the Seine protocol, is presented in Section 4, and a proof of its correctness is given in Section 5. A generalization of the Seine protocol along with a proof of its correctness are discussed in Sections 6 and 7, respectively. We end with some concluding remarks in Section 8.

2. The Problem

It is required to design a communication protocol between identical *senders* and one *receiver*. At every instant, each sender sends one bit and the receiver receives one bit. The bit received by the receiver at any instant equals

$$b_0 + b_1 + \ldots + b_{k-1}$$

where each b_i is the bit sent by the i^{th} sender at that same instant, k is the number of senders, and "+" is the boolean "inclusive or" operator. An outline of this arrangement is shown in Figure 1.

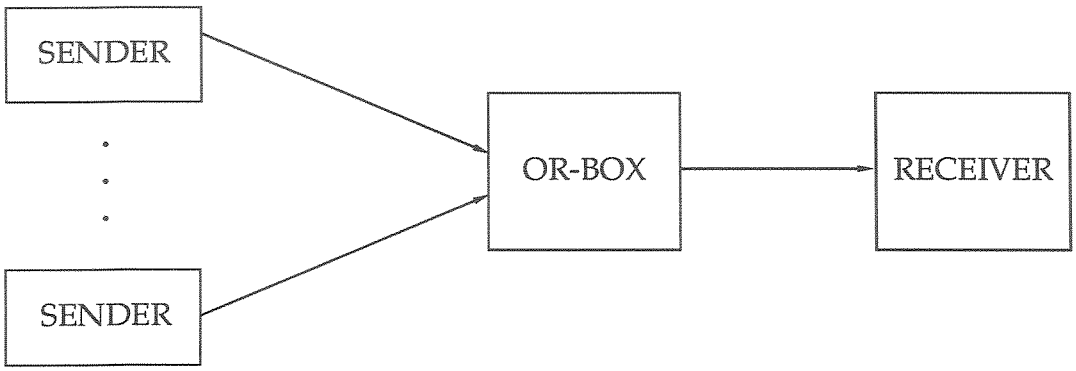


Figure 1. Block diagram of the System.

Most of the time a sender has no string of data bits to send, in which case it continues to send zero bits. At arbitrary instants, a sender gets an arbitrary, but finite, string x of data bits to send. At such instants, the sender first encodes the string x in some appropriate form called the *message of* x, then sends the bits of the message, one by one. At the end, the sender returns to sending zero bits, until it gets the next string of data bits to send, and the cycle repeats. Henceforth, the message of x is denoted $msg[x]$.

The receiver receives the bits of the incoming stream one at a time. When the receiver recognizes a contiguous block of received bits as $msg[x]$ for some bit string x, it accepts x.

The protocol between the senders and the receiver is required to satisfy the following two properties:

1. *Delivery*: If a sender sends the bits of some message $msg[x]$ while the other senders send zero bits, then the receiver will accept x.

2. *Discrimination*: If the receiver accepts a string x, then one of the senders must have sent $msg[x]$.

A design of the required protocol should define: the encoding of sent messages (i.e., define $msg[x]$ for any bit string x), and the programs for one sender and for the receiver. But before we present our design of the protocol in Section 4, we discuss in the next section some of the ideas that led us to that design.

3. Search for a Solution

In considering a solution for the above problem, it is reasonable to assume that each message $msg[x]$ has the following structure:

$$\text{head} \;.\; code[x] \;.\; \text{tail}$$

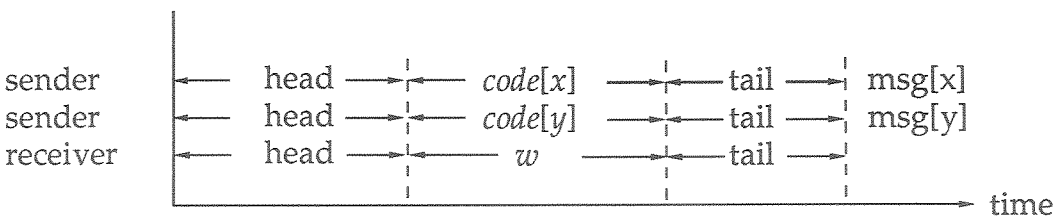
where "." : is the concatenation operator,
head : is a fixed bit pattern that declares the start of the message,
$code[x]$: is an encoding of the data string x using an appropriate coding procedure, and
tail : is a fixed bit pattern that declares the end of the message.

In order to explain our choices for "head," "$code[x]$," and "tail," we discuss a number of scenarios. In each scenario, the receiver could be fooled into accepting a data string that was not sent by any sender (thus violating the discrimination requirement), unless "head," "$code[x]$," and "tail" are chosen in a certain way.

A word of warning is now in order. The fact that some violations of particular scenarios can be avoided by certain choices of "head," "$code[x]$," and "tail" does not, in its own right, establish the correctness of these choices. The exercise merely suggests the choices. However, in order to establish the correctness of these choices, one has to prove that they always satisfy the delivery and discrimination requirements discussed earlier. A proof of the correctness of these choices is given in Section 5.

3.1 $code[x]$

Assume that two senders send two messages $msg[x]$ and $msg[y]$, where $x \neq y$, over the same time period. In this case, the receiver receives the bit-wise or-ing of the two messages during the same period. This scenario is illustrated as follows.



The heads of the two messages coincide in the same time period, and the receiver receives a head in that period. Similarly, the two tails coincide in the same period, and the receiver receives a tail in that period. Thus, the receiver ends up receiving:

head . w . tail

where w is the bit-wise or-ing of $code[x]$ and $code[y]$. If $w = code[z]$ for some z, then the receiver will accept z, possibly violating the discrimination requirement. (Notice that the discrimination requirement is not violated if $z = x$ or $z = y$.) In order to prevent this possibility, we require that the bit-wise or-ing of $code[x]$ and $code[y]$ does not generate $code[z]$ for any z.

This can be accomplished by requiring that the code of each data string be *balanced* [1], i.e. have equal numbers of 0's and 1's. In this case, if $code[x] \neq code[y]$, then the or-ing w of $code[x]$ and $code[y]$ will have more 1's than 0's. In other words, w is not balanced and so cannot equal $code[z]$, for any z.

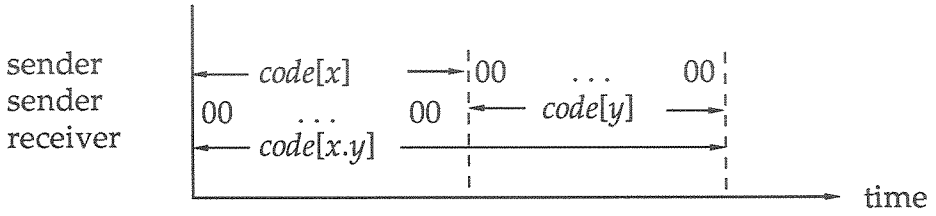
The simplest way to achieve a balanced code is to adopt *Manchester coding* [1]. In this coding, each *code* $[x]$ is constructed from its data string x as follows: replace each 0 bit in x by a combination 1.0 and each 1 bit in x by a combination 0.1. Thus,

$$\begin{aligned} code[x] &= 1.0.code[x'] && \text{if } x = 0.x' \\ &= 0.1.code[x'] && \text{if } x = 1.x' \\ &= <>, \text{ the empty string} && \text{if } x = <> \end{aligned}$$

The fact that Manchester coding has been adopted in many local area networks, e.g. Ethernets [5], makes this coding attractive for our protocol.

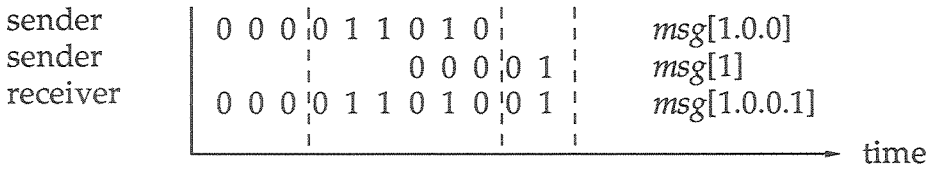
3.2 Head

To show that nonempty heads are needed, consider the following scenario in which one sender sends $code[y]$ immediately after another sender finishes sending $code[x]$.



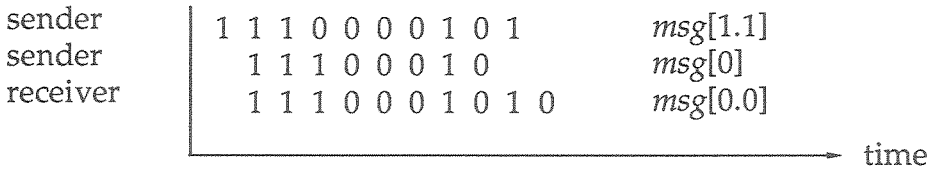
The receiver ends up receiving *code*[*x.y*] and so accepts the data string *x.y* which was not sent by either sender, violating the discrimination requirement. Therefore, a nonempty head should be provided for each message.

The head of a message should be a bit pattern that (i) does not appear in *code*[*x*] for any *x*, and (ii) cannot be created due to message collision. The simplest such pattern is "0.0.0". To show that this pattern is not enough as a head, consider the following scenario in which one sender sends *msg*[1.0.0] and another sender sends *msg*[1].



In this case, the receiver ends up receiving *msg*[1.0.0.1] which was not sent by either sender, violating the discrimination requirement. Even if one or two 1's would precede the three 0's in the message's head, the same scenario still violates the discrimination condition. Fortunately, if three 1's precede the three 0's in the message's head, this scenario no longer violates the discrimination requirement. Thus, let us consider for now that the head of each message is the bit pattern 1.1.1.0.0.0.

To show that the pattern 1.1.1.0.0.0 is not enough as a head, consider the following scenario in which one sender sends *msg*[1.1] and another sender sends *msg*[0].



In this case, the receiver receives *msg*[0.0] and accepts the data string 0.0 which was not sent by either sender, violating the discrimination requirement. If more 1's are

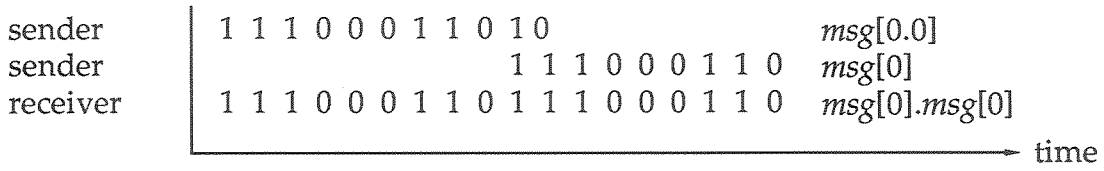
added to the left of the head, then the same scenario still violates the discrimination requirement. However, by adding only one 1 to the right of the head to become:

$$\text{head} = 1.1.1.0.0.0.1$$

then this scenario no longer violates the discrimination requirement.

3.3 Tail

To show that a nonempty bit pattern is needed as a tail for each message, consider the following scenario in which one sender sends $msg[1.1]$ and another sender sends $msg[0]$ as follows:



In this case, the receiver receives $msg[0].msg[0]$ and accepts two 0 strings, violating the discrimination requirement. This problem is solved by requiring that each message has the following tail

$$\text{tail} = 0.0$$

4. The Seine Protocol

We start our presentation of the Seine protocol by defining $msg[x]$ for any bit string x. We then define a typical sender and the receiver of the protocol.

4.1 Messages

Let $x = x_0.x_1 \ldots . x_{n-1}$ be a string of n bits. Define $msg[x]$ to be a string of $2n + 9$ bits of the form:

$$msg[x] = 1.1.1.0.0.0.1.\, y_0.y_1 \ldots . y_{2n-1}.0.0$$

where each bit x_i in string x is represented by two bits y_{2i} and y_{2i+1} in string $y = y_0.y_1. \ldots .y_{2n-1}$. Bit y_{2i} is the inverse of x_i and bit y_{2i+1} is the same as x_i. String y is usually referred to as the *Manchester code* of string x. Henceforth, we refer to string y as $code[x]$ and to string $0.1.x$ as $code[1.x]$; in particular,

$$msg[x] = 1.1.1.0.0.0.1.code[x].0.0$$
$$= 1.1.1.0.0.code[1.x].0.0$$

4.2 Sender

As specified by the problem, when the sender has no string of data bits to send, it sends zero bits. On the other hand, when the sender has a string x of data bits to send, it sends $msg[x]$ one bit at a time, starting with the three ones at the beginning of $msg[x]$.

4.3 Receiver

The program for the receiver is as follows.

```
process receiver;
   var      st : (s0, s1, s2, s3, s4) init s0,      {* see below *}
            rcvd : string of (0,1) init < >,        {* < > denotes the empty string *}
            b : (0,1),                              {* b is the received bit *}

   do forever
      receive b;
      if  st = s0 ^ b = 1  →  st := s0
      □   st = s0 ^ b = 0  →  st := s1
      □   st = s1 ^ b = 1  →  st := s0
      □   st = s1 ^ b = 0  →  st := s2; rcvd := < >
      □   st = s2 ^ b = 1  →  st := s3
      □   st = s2 ^ b = 0  →  st := s4
      □   st = s3 ^ b = 1  →  st := s0
      □   st = s3 ^ b = 0  →  st := s2; rcvd := rcvd.0
      □   st = s4 ^ b = 1  →  st := s2; rcvd := rcvd.1
      □   st = s4 ^ b = 0  →  st := s0;
                              if left-most bit in "rcvd" is 1
                              then remove this bit from "rcvd"
                                      and accept the remaining string
                              fi
      fi
   od
end receiver;
```

This program has five states: s_0 to s_4. It is convenient to attach some, rather informal, meaning to each of these states.

s_0 : no 0 bit in the current message has been received.
s_1 : the first 0 bit in the current message has just been received.
s_2 : the first 0.0 bits in the current message have been received, and the receiver is ready to receive one of the combinations 1.0, 0.1, or 0.0.
s_3 : bit 1 in the next combination of 1.0 in $code[1x]$ of the current message has just been received.
s_4 : bit 0 in the next combination of 0.1 in $code[1x]$ of the current message has just been received, or the first 0 bit of the combination 0.0 at the end of the current message has just been received.

5. Protocol Correctness

Correctness of the above protocol is established by showing that the protocol satisfies the two properties of delivery and discrimination stated in Section 2.

Theorem 1: (*Delivery*) If a sender sends the bits of some message $msg[x]$ while the other senders send zero bits, then the receiver will accept x.

Proof: Assume that a sender sends the bits of $msg[x]$ for some string x while other senders send zero bits. Then the receiver will receive the bits of $msg[x]$ unaltered. Since the bits of $msg[x]$ constitute a string in the form 1.1.1.0.0.$code[1x]$.0.0, it is sufficient to show that if the receiver is at any state of its five states, and receives a string 1.1.1.0.0.$code[1.x]$.0.0, it will accept x. The proof is divided into three parts, each of them can be checked from the above receiver program. First, if the receiver is at any state and receives a string 1.1.1.0.0, it will reach state s_2 with $rcvd = <>$. Second, if the receiver is at s_2 with $rcvd = <>$ and receives a string $code[1.x]$, it will reach s_2 with $rcvd = 1.x$. Finally, if the receiver is at s_2 with $rcvd = 1.x$ and receives a string 0.0, it will reach state s_0 after accepting x. ❑

This proof of Theorem 1 reveals an interesting, and somewhat unexpected, property of the protocol. Reception of the head bit pattern 1.1.1.0.0.0.1 starting from any state, is guaranteed to drive the receiver into a state s_2 with $rcvd = 1$. This indicates that the receiver will accept a valid message starting from any state. Thus, the proper initialization of the receiver is not critical to the correctness of the protocol in

the long run. (In the short term, however, the receiver may accept at most one invalid message, or may reject at most one valid message due to its improper initialization.) This property of the protocol is usually referred to as self-stabilization; see for instance [2], [3], and [8].

Theorem 2: (*Discrimination*) If the receiver accepts a string x, then one of the senders must have sent $msg[x]$.

Proof: We divide the proof into the following two parts.

a. If the receiver accepts a string x, then it must have received a string 0.0 $code[1x]$ 0.0.

b. If the receiver receives a string 0.0.$code[1x]$.0.0, then one of the senders must have sent $msg[x]$.

Part a: Assume that the receiver accepts a string x. Then it must have reached state s_0 with $rcvd = 1.x$, or equivalently, it must have received a string 0.0 after reaching state s_2 with $rcvd = 1.x$. This means that the receiver has received a string $code[1.x]$.0.0 after reaching state s_2 with $rcvd = <>$. This in turn means that the receiver has received a string 0.0.$code[1.x]$.0.0 (after reaching s_0).

Part b: Assume that the receiver receives a string 0.0.$code[1.x]$.0.0. Since the first two bits in this string are 0's, each sender must have sent two 0's in these positions. (Recall that the bits sent in the same position are or-ed into one bit before delivery to the receiver.) Similarly, since the last two bits are 0's, each sender must have sent two 0's in these positions.

Now what could a sender have sent in the remaining positions? Clearly, no sender could have started a new message in these positions since this would require sending 1.1.1 which would have to be received unaltered by the receiver, and this violates the fact that the receiver has received in these positions $code[1.x]$ which cannot have the string 1.1.1. Therefore, in these positions, each sender must have sent either a string 0.0...0 or a string $code[1.y]$.0.0...0, for some string y.

Without loss of generality, assume that $code[1.x]$ has $2n$ bits for some $n \geq 1$. Now if each $code[1.y]$ sent by a sender in these positions has less than $2n$ bits, then the bits sent in the last two positions are 0.0, contradicting the fact that the last two bits in

code[1.x] are either 0.1 or 1.0. Therefore, at least one of the senders must have sent in these positions some *code*[1.z] that has $2n$ bits.

It remains now to show that this *code*[1.z] is identical to *code*[1.x]. This proof is by contradiction. Assume that this *code*[1.z] is different from *code*[1.x]. This means that *code*[1.z] while being sent has collided with at least one concurrently sent *code*[1.w], and the result was *code*[1.x] different from *code*[1.z]. However, collision can only increase the number of ones in the received bits. Since *code*[1.z] is balanced, i.e., has equal numbers of zeroes and ones, then *code*[1.x] must be unbalanced which is a contradiction. Therefore, *code*[1.z] must be identical to *code*[1.x].

So far, we have established that at least one sender must have sent a string 00 *code*[1.x] 00. Then, this sender must have sent 1.1.1 previously, i.e., it must have sent *msg*[x]. ❑

6. The Seine Protocol Generalized

In the Seine protocol, each message *msg*[x] has $2n + 9$ bits, where n is the number of bits in the data string x. The dominant term of $2n$ is due to the fact that x is encoded using Manchester coding. In order to reduce this term, the data strings could be encoded in some other balanced code (i.e., one where each code word has equal numbers of zeroes and ones) that is more efficient than the Manchester code (i.e., one in which each data string with n bits can be encoded in less than $2n$ bits). Note that the Manchester code is a special instance of a balanced code.

Assume that the number of bits in every data string to be sent by a sender is a multiple of m, for some m. (This is not a restriction in case $m = 1$.) Therefore, *code*[x] for every data string x can be constructed as follows. First, partition x into words of m bits each; then replace each word with a corresponding code word from some balanced code C. If each code word in C has p bits, then each *code*[x] will have $p(n/m)$ bits, where n is the number of bits in string x. The *efficiency* of this coding scheme is m/p. In the case of Manchester coding, $m = 1$, $p = 2$, and the efficiency = 1/2. Next, we show that $m \geq 4$ is both necessary and sufficient for achieving a code that is more efficient than the Manchester code.

Because each code word is balanced, i.e., has equal numbers of 0's and 1's, p must be even. If $m = 1$ then C has two code words, and so each code word, being balanced,

has at least two bits. In this case, the resulting code is the Manchester code. If $m = 2$ then C has four (balanced) code words. Thus, each code word has at least four bits, and the resulting code has the same efficiency as the Manchester code. If $m = 3$ then C has eight (balanced) code words. Because there are only six balanced code words if we choose $p = 4$, and because p must be even, p is at least 6, and the resulting code again has the same efficiency as the Manchester code. However, if $m = 4$ then C has 16 balanced code words. Many balanced code words can be found by choosing $p = 6$. For example, the following set of 16 balanced code words is a possible C.

$C = \{$ 0.0.1.0.1.1, 0.0.1.1.0.1, 0.1.0.0.1.1, 0.1.0.1.0.1,
0.1.0.1.1.0, 0.1.1.0.0.1, 0.1.1.0.1.0, 0.1.1.1.0.0,
1.1.0.1.0.0, 1.0.0.1.1.0, 1.0.1.1.0.0, 1.0.1.0.1.0,
1.0.1.0.0.1, 1.0.0.1.1.0, 1.0.0.1.0.1, 1.0.0.0.1.1 $\}$

In this case, the efficiency of C is 4/6 much higher than the efficiency of the Manchester code. If m is larger than 4, the efficiency of the resulting code C can be higher than 4/6.

In the remainder of this section, we present a generalization of the Seine protocol for the case in which the data strings are encoded using any balanced code C.

6.1 Messages

From the given set C of code words, define the following three quantities:

i = the maximum of two numbers: the maximum number of successive ones in two adjacent code words in C, and the maximum number of successive ones at the beginning of any code word in C plus one.

j = the maximum number of successive zeroes in two adjacent code words in C,

k = the maximum number of successive zeroes at the beginning of a code word in C.

For instance, $i = 4$, $j = 4$, and $k = 2$ for the set C defined earlier.

The three quantities, i, j, and k, are used in defining $msg[x]$ as follows.

$$msg[x] = (1)^{i+1}.(0)^{j+1}.1.code[x].(0)^{k+1}$$

where $code[x]$ is the encoding of string x using the code words in set C as discussed earlier. Notice that if C is a Manchester code, then $i = 2$, $j = 2$, and $k = 1$, and $msg[x]$ reduces to the form given in Section 4.1.

6.2 Sender

As before, when the sender has a string x of data bits to send, it sends $msg[x]$ one bit at a time starting with the $(1)^{i+1}$ bits at the beginning of $msg[x]$. Otherwise, it sends zero bits.

6.3 Receiver

The program for the receiver depends on the chosen code C and its three quantities, i, j, and k. It is as follows.

```
process receiver (C, i, j, k);
   var      st : 0, 1, ..., j+2 init 0,
            rcvd,w : string of (0,1) init < >,   {* < > denotes the empty string *}
            b : (0,1),                           {* b is the received bit *}

   do forever
      receive b;
      if  st ≤ j ^ b = 1     →  st := 0
      ▯   st ≤ j ^ b = 0     →  st := st+1
      ▯   st = j+1 ^ b = 1   →  st := j+2; rcvd := < >; w := < >
      ▯   st = j+1 ^ b = 0   →  st := 0
      ▯   st = j+2           →
                                w := w.b;
                                if w = proper prefix of a code word in C → skip
                                ▯ w = code word in C → rcvd := rcvd.word(w); w := < >
                                ▯ w = 0^(k+1) → st := 0; accept rcvd
                                ▯ else → st := 0
                                fi
      fi
   od
end receiver
```

This program has j+3 states, named 0 to j+2. It uses the function $word(w)$ which returns the word that corresponds to any given code word w in C.

7. Correctness of the General Protocol

The next two theorems establish that the general protocol satisfies the delivery and discrimination properties stated in Section 2.

Theorem 3: (*Delivery of the General Protocol*) If a sender sends the bits of some $msg[x]$ while the other senders send zero bits, then the receiver will accept x.

Proof: It is sufficient to prove the following two assertions:

a. If the receiver is at any state of its j+3 states, and receives a string $(1)^{i+1}$, it will reach a state 0.

b. If the receiver is at a state 0 and receives a string $(0)^{j+1}.1.code[x].(0)^{k+1}$, it will accept x.

Part *b* follows from the receiver's program. We concentrate on Part *a*.

If the receiver is at any of the states 0, ..., j, and receives a nonempty string of 1's, then it will reach state 0. It remains now to show the same when the receiver is at state j+1 or j+2.

Case 0: Assume that the receiver is at state j+1 and receives a string $(1)^{i+1}$. After receiving the first 1, the receiver will reach state j+2 with w = < >. Then, after receiving the remaining 1's, w will become different from any prefix of any code word in C, and different from $(0)^{k+1}$. Therefore, the receiver will reach state 0.

Case 1: Assume that the receiver is at state j+2 and receives a string $(1)^{i+1}$. Then, regardless of the current value of w, concatenating as many 1's to the tail of w will make w different from any prefix of any code word in C and different from $(0)^{k+1}$. Therefore, the receiver will reach state 0. ❑

It follows from the proof of Theorem 3 that the receiver can accept a valid message starting from any state. Hence, the general seine protocol is self-stabilizing in the same way as the original protocol.

Theorem 4: (*Discrimination of the General Protocol*) If the receiver accepts x, then one of the senders must have sent $msg[x]$.

Proof: Similar to the proof of Theorem 2, we divide the proof into the following two parts.

a. If the receiver accepts a string x, then it must have received a string $(0)^{j+1}\ 1\ code[x]\ (0)^{k+1}$.

b. If the receiver receives a string $(0)^{j+1}\ 1\ code[x]\ (0)^{k+1}$, then one of the senders must have sent $msg[x]$.

The proof of each part is similar to that of the corresponding part in the proof of Theorem 2. For convenience, we state the proof for Part *b*.

Assume that the receiver receives the string $(0)^{j+1}\ 1\ code[x]\ (0)^{k+1}$. Since the first $(j+1)$ bits in this string are zeroes, each sender must have sent zeroes in these positions. Similarly, since the last $(k+1)$ bits are zeroes, each sender must have sent zeroes in these positions. Also, no sender could have started a new message in the remaining positions, since this would require sending $(1)^{i+1}$ which have to be received unaltered by the receiver, and this violates the fact that the receiver has received in these positions $1\ code[x]$ which cannot have $(1)^{i+1}$. Therefore, in the remaining positions, each sender must have sent either a string 00...0 or a string $1\ code[y]\ 00...0$, for some string y.

Without loss of generality, assume that the received $code[x]$ has k code words. Now, if every $code[y]\ 00...0$ sent by a sender in these positions has less than k code words, then the positions of the last code word in $code[x]$ must be all zeroes, contradicting the fact that a code word is balanced, i.e., has an equal number of zeroes and ones. Therefore, at least one of the senders must have sent in these positions $code[z]$ where the number of code words in $code[z]$ equals that in $code[x]$.

It remains now to show that this $code[z]$ is identical to $code[x]$. The proof is by contradiction. Assume that $code[z]$ is different from $code[x]$. This means that

code[z] while being sent has collided with at least one concurrently sent *code*[*w*], and the result was *code*[*x*] different from *code*[z]. However, collision can only increase the number of ones in the received bits, but since *code*[z] is balanced, then *code*[*x*] must be unbalanced, and we have a contradiction.

So far, we have established that at least one sender must have sent a string $(0)^{j+1}$ 1 *code*[*x*] $(0)^{k+1}$. Thus, this sender must have sent $(1)^{i+1}$ previously, i.e., it must have sent *msg*[*x*]. ❑

8. Concluding Remarks

We have presented a family of protocols for data transmission from multiple identical senders to a single receiver. One protocol from this family has been implemented on a local area network called the *D-network*, Figure 2. In this network, all stations are connected by a unidirectional fiber link. Each station has a sender and a receiver. Transmissions from all the senders are or-ed together into one bit stream traveling on the fiber link and is received by all receivers at different instants. The details of this implementation are described in [7].

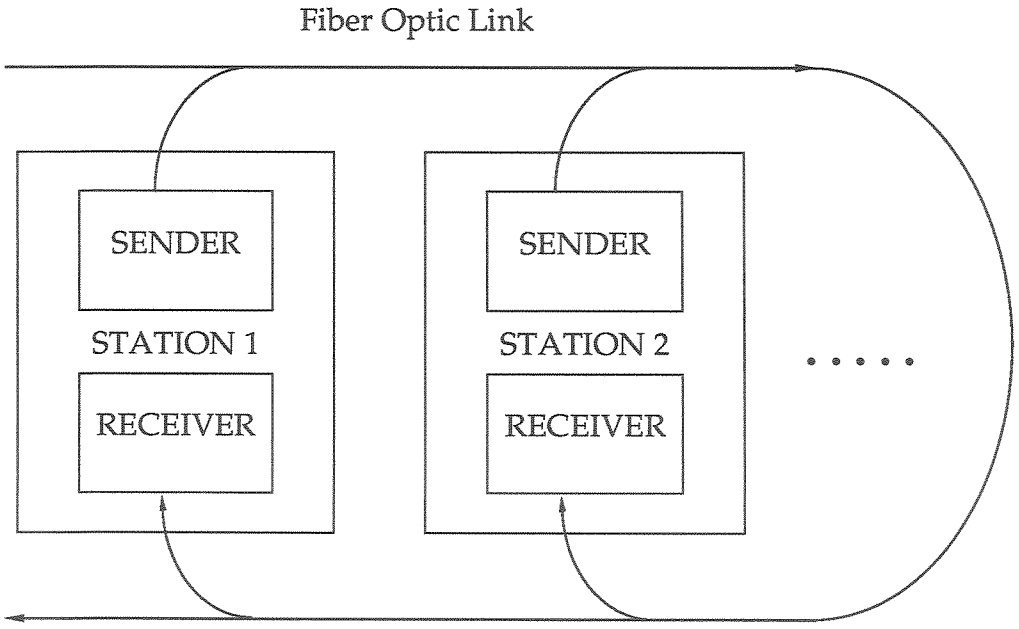


Figure 2. The D-Network.

References

[1] E. E. Bergmann, A. M. Odlyzko, and S. H. Sangani, "Half-Weight Block Codes for Optical Communications," *AT&T Technical Journal*, Vol. 65, No. 3, May-June 1986, pp. 85-90.

[2] G. M. Brown, M. G. Gouda, and C. L. Wu, "Token Systems that Self-Stabilize," *IEEE Trans. on Computers*, Vol. 38, No. 36, 1989, pp. 845-852.

[3] M. G. Gouda and N. Multari, "Stabilizing Communication Protocols," in preparation, 1989.

[4] N. F. Maxemchuk, "Random Access Strategies for Fiber-Optic Networks," Proceedings of the INFOCOM '87, pp. 307-311.

[5] N. F. Maxemchuk, "Twelve Random Access Strategies for Fiber-Optic Networks," *IEEE Trans. on Communications*, Vol. 36, No. 8, 1988, pp. 942-950.

[6] R. M. Metcalfe and D. R. Boggs, "Ethernet: Distributed Packet Switching for Local Computer Networks," *Commun. ACM*, Vol. 19, July 1976, pp. 395-404.

[7] U. Mukherji, N. F. Maxemchuk, C. W. Wu, and J. C. Swartzwelder, "Transmission Format and Receiver Logic for Random Access Strategies in a Fiber-Optic Network," presented at *IEEE Computer Networking Symposium 1988*, Washington, DC, April 11-13, 1988.

[8] N. Multari, "Toward a Theory for Self-Stabilizing Protocols," Ph.D. Dissertation, Dept. of Computer Sciences, Univ. of Texas at Austin, July 1989.