Bootstrap Network Resynchronization

(Extended Abstract)

Yehuda Afek
Computer Science department
Tel-Aviv University
Israel 69978.

Eli Gafni*
Department of Computer Science,
University of California, Los Angeles,
and Tel-Aviv University.

Abstract

The problem of applying static distributed algorithms in an *eventually connected* network is addressed. We propose a new technique, *bootstrap resynchronization*, by which any static diffusing computations can be reliably adapted to such a network. The communication overhead for each execution of the diffusing computation adapted by our technique is $O(nE)$ messages, were n and E are the total number of nodes and links in the network. The time overhead of the bootstrap resynchronization per execution is $O(n^2)$.

One of the simplest diffusing computations is *flood.* Flood can be used to send a data item in the end-to-end problem, and to receive its acknowledgment in a return flood. Applying our technique to this algorithm results in an $O(nE)$ messages per data item, data oblivious end-to-end protocol ($O(E \cdot D + nE \log n)$ bits of communication per data item of size D bits).

*Supported by NSF Presidential Young Investigator Award under grant DCR84-51396 & matching funds from XEROX Co. under grant W881111.

1 Introduction

In this paper we introduce and present a new general scheme, the *bootstrap resynchronization*, to reset any (static or dynamic) diffusing computation in the *eventual* network model. Reset here means to purge the effects of old executions and to start a new one without being confused between the executions. The power of our scheme is exemplified by using it to produce an $O(nE)$ messages solution to the end-to-end communication problem in *eventually connected* networks, where n and E are the total number of nodes and links in the network, respectively. (The best previous solution has complexity $O(n^9)$ [AMS89].)

The *eventual* network model considers dynamic data communication network in which links may fail and recover infinitely often. In this model the *eventual network* consists of all the links which are not down forever. Two nodes that belong to the same fragment of the eventual network, are *eventually connected*, and an eventual network which consists of a single fragment is an *eventually connected* network. We address the problem of resynchronizing a diffusing computation relative to such a network so that it accomplishes its required task.

For instance, assume a query is issued at a site of a distributed directory system. Assume the query requires certain quorum of sites to agree. If such a quorum exits within the fragment that is eventually connected to the issuing site, there is no reason why the query should not eventually return with an answer. Of course in practice one may abort the query if it does not return within some prescribed time-out, but it is satisfying if the liveness of the query does not depend on time or on an assumption that there exist a network snapshot that contained the quorum. Resynchronization provides for the safety of the query. Messages of the current query should not be confused with messages of possibly the same query issued (and terminated or aborted) long ago.

In a diffusing computation, one node, the originator, receives a trigger signal from the environment, after which the originator may send messages to its neighbors. Following a reception of a message a node may send messages to its neighbors. In this manner the trigger generates a diffusing computation in the network [DS80]. Whenever the originator receives a new trigger a new diffusing computation is started. The task of resynchronization is to distinguish at each node the messages of the most recent diffusing computation from previous ones, facilitating the uninterrupted execution of the most recent computation.

A problem that is easily solved by diffusion computation is the End-to-end communication problem: In this problem there are two distinguished nodes in the network, the *sender* and the *receiver*. The sender S has an input tape with an infinite sequence $I = (D_0, D_1, \ldots)$ of data elements. S reads these data elements and tries to transmit them to the receiver R. The receiver R must write these data elements onto an output tape O. The goal is to design a protocol such that, (a) *safety:* O is always a prefix of I and (b) *liveness:* if the sender and the receiver belong to the same fragment of the eventual network then every data element D_i is eventually in O.

Resynchronization is used to provide end-to-end service as follows: Each data item is sent to the receiver by a diffusing computation consisting of a flood of the data item from the sender to the receiver, followed by a flood of an acknowledgment from the receiver to the sender. Upon receiving the acknowledgment, the sender triggers the bootstrap resynchronization procedure and starts a new diffusing computation to deliver the next data item, etc.

More interesting is the observation, shown later, that end-to-end service may be used to provide resynchronization service. The crux of the paper is to employ resynchronization in conjunction with end-to-end communication. Resynchronization facilitates end-to-end communication between resynchronized nodes. End-to-end communication between resynchronized nodes facilitates the resynchronization of new nodes, hence the scheme name *bootstrap resynchronization*.

In another application of our scheme, Attiya Bar-Noy and Dolev used it to simulate read/write registers in a dynamic network [ABND90].

1.1 Related work and simple "unbounded" resynchronization procedures

A simple network-resynchronization algorithm can be designed by enumerating the triggers consecutively and inductively attaching these numbers to messages of each diffusing computation. Furthermore, if FIFO behavior of the links is assumed, only the first message of each trigger on a link has to carry the trigger number. The trigger number of the succeeding messages is implied. In both cases messages have to carry a theoretically unbounded overhead field. The message complexity of such resynchronization algorithms is theoretically unbounded [AAG87, Fin79].

Alternatively, another simple network-resynchronization algorithm can be designed by keeping track at each node of the number of "first-messages" it receives over each link. Such a solution, which essentially passes time-stamps by counting in some fixed quanta, is also of unbounded complexity.

Depending upon the fairness conditions at hand, different resynchronization algorithms of bounded message complexity have been proposed. The first attempt at a bounded message complexity resynchronization algorithm was reported by Finn [Fin79], who also suggested and named the problem. The proposed algorithm was found to be faulty [SH86].

The first correct algorithm [AAG87] was given under the fairness condition that nodes never fail and each message on a link eventually is either delivered, or instead a "fail" message is delivered at both ends of the link. The problem solved by [AAG87] was restricted in the sense that it did not require distinction between new and old messages if any topological change occurred after the last trigger. Only the computation triggered last, whose messages did not encounter any topological change, is required to be correct. Thus in an environment in which the network does not stabilize (which will be the case if the network is large enough), [AAG87] does not require progress. The algorithms in [AG88] imply that resynchronization can be achieved assuming no liveness conditions for specific links; that is, resynchronization at bounded cost is possible in eventually connected networks. The algorithm in [AG88] is of exponential message complexity.

A surprising and novel end-to-end algorithm under the same fairness conditions was then exhibited in [AMS89], which resulted in polynomial $O(n^9)$ message complexity. In [AGR90] some of the ideas of [AMS89] were combined with those of [AFWZ88] to derive a simpler polynomial algorithm whose message complexity is $O(n^2E)$. In addition these ideas together with [Rab89] resulted in a linear amortized message complexity ($O(n)$) end-to-end protocol.

Here, by introducing the notion of resynchronization of eventual networks and devising an efficient scheme to implement it, we improve on [AMS89] and on the worst case of [AGR90] by exhibiting an $O(nm)$ message complexity resynchronization algorithm, where n and m are the set of nodes and links, respectively, of the sub-net over which messages of the diffusing computation have propagated. We then use this solution to exhibit an $O(nE)$ message complexity algorithm for the End-to-end communication problem, where n is now the size of the node set of the whole network and E is the size of the link set.

Our resynchronization scheme is data oblivious, i.e. it treats the data item as a string of bits to which one can append a header and which one can ship from node to node, but the data string itself cannot be broken, read, or interpreted. Interestingly, in [AGR90] it is shown that for non-oblivious techniques, if data items are large enough ($\Omega(n^2 \log n)$ bits) then an $O(nD)$ bits communication complexity solution exists, where D is the size of a data item in bits.

2 Model

The eventual network notion is an abstract notion handy for defining liveness requirements. For describing algorithms an equivalent notion of ∞-*delay model*, was introduced in [AG88, AE86]. This model is the standard asynchronous point-to-point network model [Gal82, Seg83], except that the delay incurred by messages going over a link may be infinite. (While in the standard model a message may incur an unbounded but finite delay.) The ∞-delay model is thus weaker than the standard model, in that the standard model is a special case of it.

The possibility of infinite delay requires that algorithms never rely on the reliability of a single link. For instance, consider the problem of broadcast. In the standard network model it is enough to send messages over a spanning tree. Although there is no bound to the time of the broadcast, eventually all nodes will receive it. That is not the case in the infinite delay network. Thus to achieve a reliable broadcast in an infinite delay network, a flood should be used.

The ∞-delay model is a relaxation of the *dynamic network* model. A detailed and formal specification of the dynamic network model can be found in [AAG87]. In [AG88] it is shown that the dynamic network can be reduced to an ∞-delay model (in the sense that any solution in the ∞-delay model is a solution in the dynamic network model) by implementing an alternating bit protocol over each link of the dynamic network. Essentially, whenever a node wants to pass a message over an adjacent link it hands that message to the corresponding link level protocol [BSW69, HZ87]. The link level protocol is guaranteed to pass the message over the link to the node at the other end, if the link eventually comes up. Thus, a link which eventually goes down and never comes up again corresponds to an ∞-delay link (recall that nodes do not fail, i.e., never lose their memory). Hence, in our model each link has a *finite* capacity in the sense that only a constant number of messages is allowed to be in transit on a given link at a given time. Links are also assumed to obey the FIFO order on message arrivals.

2.1 Complexity measures

The *message complexity* of the resynchronization problem is the number of control messages sent in the network per originator trigger. More precisely, it is the worst case number of messages transmitted in the period of time between any two successive originator trigger events. The *time complexity* of a resynchronization algorithm is the maximum time from a final trigger until the whole network is synchronized, under the assumption that link delays are bounded by one time unit, for all the eventually operational links (also called viable links). The *space complexity* of an algorithm is the maximum number of bits, per incident link, required at any node of the network. In the End-to-end algorithm each message may carry a data element and $O(\log n)$ bits. The message complexity of the End-to-end communication is analogously defined, except that transmission of a new data item takes the place of the trigger event.

3 The resynchronization algorithm

3.1 Informal description

Consider the following problem: perform one diffusing computation in a network over which no diffusing computation has yet been executed. Clearly such a computation requires no resynchronization, any message received by a node is a message of the only diffusing computation that ever runs in the network. The basic idea of our resynchronization scheme is that for each new diffusing computation a new "copy" of the network is reconstructed around the originator, and the computation is run over that clean copy as if that is the only computation ever executed on the network. Thus, upon receiving a trigger for a new diffusing computation, the algorithm detaches the originator from the current copy to which it belonged, resets the state of the originator to the initial state, and defines the originator to be the first node in a new copy of the network. The scheme proceeds to grow the new copy by expanding over links and nodes adjacent to the new copy. An invariant maintained by the scheme is that at any given time any node or link belongs to at most one copy of the network. An important property of our scheme is that only the viable copy, the one to which the originator belongs, will have the ability to expand over new nodes and links taken away from obsolete copies of the network.

Let us first describe an unbounded implementation of the above idea. It is based on an unbounded ordered set of labels used to uniquely tag each resynchronized diffusing computation. Whenever the originator receives a trigger it chooses a new label, *s-new* which is larger than any that has been previously used. It sets its *label* to *s-new* and performs the following resynchronized broadcast to disseminate this new label to all the nodes in the network. Each node, upon receiving a new label a that is larger than any it has received so far, sets its *label* to a and resets its state for a new diffusing computation. It then sends its *label* to all its neighbors.

Thus, time at each node consists of intervals, each of which is associated with a *label*. Moreover at any point of time each node i has a $view_i(j)$ about the interval *label* of each of its neighbors j. As we will see later, we require that each time node v changes its label it has to update the view of its neighbors label, to a label that the neighbor had after the change. Let link (i,j) be *coordinated* at i at time t, if at that time the label of i agrees with $view_i(j)$.

All the intervals at a node are labelled uniquely, and moreover if no new trigger arrives, eventually the labels of all the nodes in the eventual network will agree with that of the originator. It is now easy to see that resynchronization is achieved if nodes consider themselves involved in a new diffusing computation upon changing labels, and then receive computation messages only over coordinated links (discarding all messages received over non-coordinated links), and delay sending computation messages on a link until it is coordinated. Note that a necessary assumption is that the links are FIFO, thus ensuring that all the computation messages following a synchronized broadcast belong to the corresponding diffusing computation.

There are three important properties guaranteed by the availability of a brand new label when a new trigger arrives, that is larger than all previous labels: (1) There is no node in the network with this label, (2) the most recent label is easily identified as such, (3) No message that is associated with this label is currently in transit on any link in the network.

Note that property (3) does not follow from properties (1) and (2). In case that property (3) does not hold, an old message could remain on a link and pop out in undesirable times. To overcome this, we redefine a link (i,j) to be coordinated at i at time t if at that time the label of i agrees with $view_i(j)$ and moreover the interval at j associated with $view_i(j)$ overlaps in time the current interval at i. This is readily checked by implementing a token over a link, sending it with a label and receiving it back with the same label while at the same interval. That is, the token is used to implement a "read." The same token is used in all such checkes in each direction.

We saw how resynchronization can solve the end-to-end problem, in the following we show how resynchronization with bounded labels is achieved by end-to-end. Properties (1) and (2) are obtained by implementing the resynchronized broadcast by n parallel and independent end-to-end communication protocols between the originator and the nodes of the network. The end-to-end communication between the originator and node v is used to inform node v of the originator's new label. In each such end-to-end dialogue the sender holds the *current* message to be sent, and an empty place for the *next* message to be sent. When the current message is acknowledged by node v there are two possibilities, either the message that was just acknowledged was an old label and the newest label should be sent to v next, or the acknowledged message contained the most recent label and no new label is available for broadcasting. In the former case, as soon as the current message is acknowledged, the empty place of the next message is filled by the new label (there could be some labels between the one acknowledged and the new one, that the originator never sends to v). In the latter case, the originator waits until the next message is available. Since in the end-to-end dialogue we do not send the next message until the previous one has been acknowledged it follows that when the need for a new label arises, the uncertainty of the originator with respect to the label of any node is at most between the current label being sent to the node and the one that preceded it. Thus, by the pigeon hole principle, the availability of $2n - 1$ labels is sufficient to allow the originator to choose a new label satisfying properties (1) and (2).

All that is left, is to implement the end-to-end communication between the originator and each node. We do not need to be able to proceed with the end-to-end communication with node i as long as none of its neighbors received the most recent label. That is, it is enough that end-to-end will proceed only with nodes which are at the periphery of the resynchronized part of the network to which the originator belongs, and which inductively will be a connected subgraph. Moreover, the number of data items that have to be transmitted between the originator and a node in the periphery, via the resynchronized territory is at most two, one containing an obsolete label and one which contains a label that is associated with the resynchronized territory.

The Alternating Bit Protocol [BSW69] establishes end-to-end communication when the sender and receiver are connected by two, possibly lossy, fifo directed channels. Thus the problem reduces to establishing a fifo (lossy) communication between the origin and nodes in the periphery of the resynchronized part of the network. We view a connected set of nodes and links that are synchronized and coordinated by a label, as a virtual link. The transmission of messages over the virtual link is implemented by a flood mechanism in the corresponding set of nodes and links. To send a message M from node u to node v, u simply initiates a flood of the message (M, v) over the resynchronized part. Furthermore, each such virtual link by itself can be made fifo via the use of sequence numbers, since as we have already shown the number of end-to-end messages over a single virtual link is bounded (two per node). The originator is connected to a single virtual link, namely the links and nodes synchronized to its label. When node j adjacent to i assumes a new label the virtual link of the label of j is connected to i via link (i, j). The originator and node i try to communicate via the virtual links between the originator and the neighbors of i, and from there through the links between i and its neighbors.

Notice that the origin is always connected to the virtual link most recently created and therefore messages received by the originator over the virtual link were not yet sent when the link was just initiated. Thus, messages of the same type which the originator receives over different virtual links, over time, obey fifo order. Hence, messages from node i received by the originator are received in the order sent (i.e., a lossy FIFO channel from each node to the originator). Establishing a FIFO virtual link from the originator to each node i is a little trickier. The main problem is to be able to order messages received at i from neighbors holding different labels. To solve this problem we rely on the fifo property of the virtual link from i to the originator. Instead of the originator "sending" messages to i, we will have i "read" the originator, that is the originator will send a message to i only in response to a message from i. That is, i will send a question to the originator by flood over a resynchronized territory, adjacent to i, and the originator will respond with a message appended with the associated question over the same label territory. That is, with every message from the originator received at i we will have the associated message from i, that generated it.

We notice the important fact: Two messages from the originator, received at i through neighbors with different labels must have been sent by the originator in the order of the messages that generated them. That is the order of messages received at i from the originator is determined by the order in which the associated questions were sent to the originator. This is because (1) once the originator detaches itself from a synchronized part of the network it never reconnects itself to that part again, and (2) if i receives a response to a question then the originator must have been connected to the part through which the response arrived, when the question was sent out of i. This leads us to the conclusion that for i to determine whether a message it receives is out of order it is sufficient for it to maintain the order of the associated generating messages (questions).

To implement the above, when node j neighbor of i changes its label to l_j it notifies i of the change. Node i then sends a message to j to be forwarded to the originator by a flood over the links synchronized by label l_j to generate a response from it. This message propagates by a flood. This solution has one problem, for each incident link, i could perform a separate flood to the originator. This would result in an $O(E^2)$ messages algorithm. To alleviate this problem, messages of i that get to the same label l_j nodes through different links are considered indistinguishable for flood purposes, so node i will cause only a single flood per message type within the territory labelled by l_j. Yet invisible to the flood, each message of the flood carries the name of the link incident to i through which it entered the l_j labelled area. For each message type, each node propagates only the first flood message of this type it receives together with the name of the link through which that message entered the labelled area. The response from the originator will carry along this link name so that when it arrives back to i, the generating message is determined, and the order on the received messages is deduced. Therefore, all that is needed is to maintain at i, is an order among its adjacent links. The place of a link in the order is in turn determined by the last time a question to the origin was made over the link.

Several optimizations are possible:

1. By alternating in the end-to-end communication between real label and a dummy label, one which the originator never assumes, we can reduce the number of distinct labels required to n instead of $2n-1$. This is because the uncertainty about the label of a node will always be between the dummy label and one good label.

2. Messages from the node to the origin can piggyback on the message that is sent to solicit response from the origin. Thus all the communication that has to take place is at most two rounds of messages: Starting at the node acknowledging the last message and carrying back the cur-

rent message, which may contain a previous label that was used by the originator, then getting it acknowledged and sending the next label, which is the current label of the originator.

3. Since we know that no two consecutive messages are the same we can determine the "parity" of the messages in the Alternating Bit protocol by looking at its content, thus we can do away with the bit overhead of the Alternating Bit Protocol.

The code of the algorithm is given in the next section.

Remarks:

1. The bootstrap resynchronization procedure can be easily made local, i.e., to run only over links that are required by the diffusing computation. This is achieved by synchronizing a link only if the diffusing computation tries to send a message over it, or if the synchronization is initiated by the other end of the link.

2. Note that the originator does not have to know the names of the nodes before the procedure starts. It may learn them as it gets the first message from each node.

3. To perform the end-to-end communication between a sender and a receiver either the sender or the receiver could assume the roll of the originator. Thus the algorithm could be receiver driven as well.

4. In the full paper we define and discuss the self-stabilization aspects of this algorithm.

4 The code

In this section we formally state the code of the protocol. The presentation of the code is based on the language of *guarded commands* of Dijkstra [DF88] where the code of each process is of the form $G_1 \rightarrow A_1 \Box G_2 \rightarrow A_2 \Box \ldots G_l \rightarrow A_l \Box$. The code is executed by repeatedly selecting an arbitrary i from all guards G_i which are true and executing A_i. A guard G_i is a conjunction of predicates. The predicate **Receive** M is true when a message M is available to be received. If the statements associated with this predicate are executed, then prior to this execution the message M is received. The message may contain some values that are assigned, upon its receipt, to variables stated in the **Receive** predicate.

The code in an ordinary node is given in Figures 1 and 2. The code for the originator which is also the sender in the end-to-end protocol, is given in Figures 2 and 3. The code of the receiver is the same as that of an ordinary node with an addition that is part of Figure 2.

The messages of the protocol are divided into two groups, flood messages and non-flood messages. The non-flood messages are **New_Label**(l) and **Ack**(l) and the flood messages are: { **ACK, D,** (Q, v, u)**,** (Q, v, u, M)} where $l, Q, M \in$ set of labels, $u, v \in$ set of nodes, and **D** is a data element.

Each node has the following variables: **coord**(v, u) which is true at v only if v has received an indication that the label at u has been the same as the present label at v. **Wait_ack**(u) tells whether the most recent New_Label message sent from v to u has been acknowledged or not. The code allows at most one New_Label message from v to u (as if it is a token), thus, before sending a new New_Label message an acknowledgment has to come back indicating that the previous one has arrived at u. **Obsolete**(u) is another boolean variable that is true if v has changed its Label after a New_Label message was sent to u and before an ack was received for that message. **Okay**(v, u) is true if the most recent New_Label message has been acknowledged (i.e., if the "token" is not in transit).

4.1 Correctness proof

Omitted from this extended abstract.

4.2 Complexity analysis

Theorem 1 *The message complexity of the algorithm is $O(nE)$ messages.*

Sketch of proof: By the end-to-end algorithm , following the last trigger at the origin each node may assume at most two more labels. The current label being sent in the end-to-end, and the next one which is the current label of the originator. Thus, following the last trigger at the origin, a link may be synchronized to at most 3 distinct labels. While synchronized, it can be observed that a node passes a constant number of messages, per node per label. Since the number of distinct labels a node may assume is 3, we obtain that this traffic over all links is $O(nE)$. While the link is not synchronized we have a constant number of messages traversing the link, per a single change in the label of any of its end nodes. Since this number of changes is bounded by a constant, we obtain that this traffic is $O(E)$. Thus the total message complexity is $O(nE)$.

It is easily seen that the time complexity of the algorithm is $O(n+d^2)$ where d is the diameter of the network and the space complexity is $O(n)$, where a word length is $O(logn)$.

Note that the End-to-end communication complexity is the communication complexity of the resynchronization procedure plus $O(E)$ data item messages.

5 Conclusion

We have defined the resynch problem for eventually connected network and shown an efficient solution to it that can be used to solve the End-to-end problem at a bearable cost.

Furthermore, we suspect that $O(nE)$ is also a lower bound to the problem. In fact, even if we allow sequence numbers, the message complexity of resynchronizing by flood remains $O(nE)$, yet the amortized complexity goes down to $O(E)$.

References

[AAG87] Y. Afek, B. Awerbuch, and E. Gafni. Applying static network protocols to dynamic networks. In *Proc. of the 28th IEEE Annual Symp. on Foundation of Computer Science*, pages 358–370, October 1987.

[ABND90] H. Attiya, A. Bar-Noy, and D. Dolev. Sharing memory robustly in message-passing systems. In *Proc. of the Ninth Annual ACM Symp. on Principles of Distributed Computing (PODC)*, pages 363–375, August 1990.

[AE86] B. Awerbuch and S. Even. Reliable broadcast protocols in unreliable networks. *NETWORKS*, 16:381–396, 1986.

[AFWZ88] H. Attiya, M. J. Fischer, D. Wang, and L. D. Zuck. Reliable communication using unreliable channels. Manuscript, 1988.

[AG88] Y. Afek and E. Gafni. End-to-end communication in unreliabel

```
Initially ⟶              ∀u neighbor of v do
        coord(v,u):=wait_ack(u):=Obsolete(u):= Okay(v,u):= false ;
                         ∀M ∈ Msgs_Type Sent_on_u(M) := false ;
        Buffer := empty ;          queue := empty ;
        Label := 0 ;
□
Receive New_Label(l) on (v,u) ⟶                          {* A message by which
                                              neighbor u tells v of its new label, l *}
        Send Ack(Label) on (v,u) ;
        If (l = Label) ∧ (¬ wait_ack(u))
        then coord(v,u) := true ;
        else coord(v,u) := false ;
        ∀M ∈ Msgs_Type Sent_on_u(M) := false ;
                         remove from queue all Ask(*,u) entries ;
        call Ask_Origin(Label,v,u) ;
□
Procedure Ask_Origin(Q,v,u):
        Send (Q,v,u) on (v,u);
        Sent_on_u(Q) := true ;
        Enqueue(Ask(Q,u),queue) ;
□
Receive (Q,v,w,M) on (v,u) ⟶
        If (Ask(Q,w) ∈ queue)
        then Remove Ask(Q,w) and entries ahead of it from queue ;
             if (Label = Q)
             then call Resynch(M) ;
        If ¬ Sent_on_u(Label) then call Ask_Origin(Label,v,u) ;
□
Procedure Resynch(M):                          {* v is labeled by a new label, M *}
        Label := M ;
        Buffer := empty ;
        ∀u neighbor of v do
                 coord(v,u) := false ;
                 Okay(v,u) := false ;                    {* Indicates whether an ack
                       {*was received for the most recent New_Label. Initially false*}
                 if Wait_Ack(v,u) then Obsolete(u):= true ;
                 else Send New_Label(M) on (v,u) ;
                      Wait_Ack(v,u):= true ;
                 ∀M ∈ Flood_Msg_Type do Sent_on((v,u),M) := false ;
□
```

Figure 1: Part 1 of the code.

Receive Ack(l) **on** $(u, v) \longrightarrow$ {* Neighbor u acknowledges the reception of New_Label from v*}

```
    if Obsolete(u)
    then Obsolete(u):= false ;
         Send New_Label(Label) on (v,u) ;
    else Okay(v,u) := true ;
         Wait_Ack(v,u) := false ;
         if l = Label then coord(v,u):= true ;
```

□

$M \in$ **Buffer** $\wedge \neg$ Sent_on$((v, u), M) \wedge (coord(v, u) \vee M = (*, u, *, *)) \longrightarrow$

```
    Send (M) on (v,u) ;        {* M is a message in the buffer
                               {*which has to be flooded over link (v,u) *}
    Sent_on((v,u),M) := true ;
```

□

Receive (M) **on** $(u, v) \wedge M \neq (*, v, *, *) \wedge$ **coord**$(v, u) \longrightarrow$ {* M is any flood message different than $(*, v, *, *)$. It arrives over a synchronized link, i.e. a message of the resynchronized flood *}

```
    if (∀M̄ ∈ Buffer ¬same(M̄,M)) then Buffer := Buffer ∪ {M} ;
```

□

Receive $(M = (Q, u, v))$ **on** $(u, v) \wedge$ **Okay**$(v, u) \longrightarrow$ {* M is a message originating at neighbor u, and M has to be forwarded, even if coord(v, u) is false *}

```
    if (∀M̄ ∈ Buffer ¬same(M̄,(Q,u,v))
    then Buffer := Buffer ∪ {(Q,u,v)} ;
```

□

Function **same**$(M_1,\ M_2)$ {* two messages that differ only in the last field are considered the same *}

```
    if (m1 = m2) or (∃a, b, c, d such that m1 = (a,b,c) and m2 = (a,b,d))
    then return(true)
    else return(false)
```

□

ACK $\notin$ **Buffer** $\wedge$ **D** $\in$ **Buffer** $\wedge$ v **is the receiver** $\longrightarrow$

```
    Output(D)                  {* Performed only at the receiver, the real
                               End-to-end delivers data item D to the environment.*}
    Buffer := Buffer ∪ {ACK} ;
```

□

Figure 2: Part 2 of the code.

Initially $\longrightarrow$ $\forall u$ neighbor of o do
coord(o,u):=wait_ack(u):=Obsolete(u):= Okay(o,u):= false ;
$\forall M \in$ Msgs_Type Sent_on_u(M) := false ;
Label := 0 ;
$\forall n \in NODES$ do Current(n) := 0 ; {*NODES is the set of all nodes*}
Buffer := {ACK} ;
$\Box$
Receive New_Label(l) **on** (o,u) $\longrightarrow$ {* A message by which neighbor u tells o of its new label, l *}
Send Ack$(Label)$ on (o,u) ;
If $(l =$ Label$) \wedge (\neg$ wait_ack$(u))$
then coord(o,u) := true ;
else coord(o,u) := false ;
$\Box$
ACK $\in$ **Buffer** $\longrightarrow$ {* The End-to-end starts a new resynchronization at the originator to deliver Data item D to the receiver*}
D := Next_Data_Item ;
Label := Select_new_label ;
$\forall n \in NODES$ Next(n) := Label
Buffer := {(D)}
$\forall u$ neighbor of o do
coord(o,u) := false ;
Okay(o,u) := false ; {* Indicates whether an ack {*was received for the most recent New_Label. Initially false*}
if Wait_Ack(o,u) **then** Obsolete(u):= true ;
else Send New_Label$(Label)$ on (o,u) ;
Wait_Ack(o,u):= true ;
$\forall M \in$ Flood_Msg_Type do Sent_on$((o,u),M)$:= false ;
$\Box$
$(Q,v,u) \in$ **Buffer** $\wedge$ $(Q,v,u,M) \notin$ **Buffer** $\wedge$ **Current**(v) $= Q$ $\wedge$
Next(v) $= M \neq$ **Null** $\longrightarrow$ {* An internal action, by which the originator replies on the virtual link to v *}
Previous(v) := Current(v) ;
Current(v) := Next(v) ;
Next(v) := Null ;
Buffer := Buffer $\bigcup$ (Q,v,u,M)
$\Box$
Procedure **Select_new_label**
Return(m) such that
$\forall u \in NODES$ Current$(u) \neq m$ and Previous$(u) \neq m$
$\Box$

Figure 3: Originator code is this and figure 2.

networks. In *Proc. of the Seventh ACM Symp. on Principles of Distributed Computing*, pages 131–148, August 1988.

[AGR90] Y. Afek, E. Gafni, and A. Rosen. Optimal fault tolerant file transfer protocol. Extended Abstract, April 1990.

[AMS89] B. Awerbuch, Y. Mansour, and N. Shavit. Polynomial end to end communication. In *Proc. of the 30th IEEE Annual Symp. on Foundation of Computer Science*, pages 358–363, October 1989.

[BSW69] K. A. Bartlett, R. A. Scantlebury, and P. T. Wilkinson. A note on reliable full-duplex transmission over half-duplex links. *Comm. of the ACM*, 12:260–261, 1969.

[DF88] E. W. Dijkstra and W. H. J. Feijin. *A Method of Programming.* Addison-Wesley, 1988.

[DS80] W. Dijkstra and C. S. Scholten. Termination detection for diffusing computations. *Information Processing Letters*, 11-1:1–4, August 1980.

[Fin79] S. G. Finn. Resynch procedures and fail-safe network protocol. *IEEE Trans. on Comm.*, COM-27:840–845, June 1979.

[Gal82] R. G. Gallager. Distributed minimum hop algorithms. Technical Report LIDS-P-1175, M.I.T. Lab for Information and Decision Systems, January 1982.

[HZ87] J. Y. Halpern and L. D. Zuck. A little knowledge goes a long way: Simple knowledge-based derivations and correctness proofs for a family of protocols. In *Proceedings of the ACM Symp. on Principles of Distributed Computing*, August 1987.

[Rab89] M. O. Rabin. Efficient dispersal of information for security, load balancing, and fault tolerance. *Journal of the ACM*, 36(2):335–348, 1989.

[Seg83] A. Segall. Distributed network protocols. *IEEE Trans. on Information Theory*, IT-29(1), January 1983.

[SH86] F. R. Soloway and P. A. Humblet. On distributed network protocols for changing topologies. Technical report, MIT, May 1986. Report LIDC-P-1564 MIT Laboratory for Inf. and Dec. Sys.