AN ASSERTIONAL CORRECTNESS PROOF OF A DISTRIBUTED ALGORITHM*

Leslie LAMPORT

Computer Science Laboratory, SRI International, Menlo Park, CA 94025, U.S.A.

Communicated by K. Apt
Received May 1982
Revised December 1982

Abstract. Using ordinary assertional methods for concurrent program verification, we prove the correctness of a distributed algorithm for maintaining message-routing tables in a network with communication lines that can fail. This shows that assertional reasoning about global states works well for distributed as well as nondistributed algorithms.

1. Introduction

Over the past half dozen years, a significant body of theory and experience in concurrent program verification has emerged [5, 9, 11]. We have learned that even the simplest concurrent algorithms can have subtle timing-dependent errors, which are very hard to discover by testing. We can have little confidence in such an algorithm without a careful proof of its correctness.

Most computer scientists find it natural to reason about a concurrent program in terms of its behavior - the sequence of events generated by its execution. Experience has taught us that such reasoning is not reliable; we have seen too many convincing proofs of incorrect algorithms. This has led to assertional proof methods, in which one reasons about the program's state instead of its behavior. Unlike behavioral reasoning, assertional proofs can be formalized - *i.e.*, reduced to a series of precise steps that can, in principle, be machine-verified.

Because assertional methods involve reasoning about the global program state, there is a widespread belief that they are not suitable for distributed programs - a belief that has been encouraged by our own prior work [4]. In this paper, we refute that belief by showing that ordinary assertional methods, developed for nondistributed concurrent programs, work well for distributed programs. We give an assertional correctness proof of an algorithm, due to Tajibnapis [12], that maintains message-routing tables for a network in which communication links can fail and be repaired. Although we recommend reading [12] for comparison, our presentation is self-contained.

* This work was supported by the National Science Foundation under grant MCS8104459.
The algorithm is described in Section 2, and an informal correctness proof is given in Section 3. Section 4 considers the problem of writing a formal proof, and one such proof is sketched in the Appendix. In the Conclusion, we discuss why assertional methods work for distributed programs.

2. Informal development of the algorithm

We assume a network of computers communicating over two-way transmission lines. We wish to devise a routing algorithm by which each computer can send messages to any other computer in the network, and each message travels over the smallest possible number of transmission lines. As an example, consider the network of Fig. 1. Messages sent by computer $B$ to computer $E$ should go via computer $C$, not via $A$ and $D$.

As we see, the network of computers can be represented by a graph in which each computer is a vertex and each transmission line is an edge. We must route messages along minimal-length paths, where the length of a path is the number of edges it contains.

Let $\delta(b, c)$ denote the distance between vertices $b$ and $c$—i.e., the length of the shortest path joining them, or $\infty$ if there is no such path, with $\delta(b, b)$ defined to be 0. We say that two vertices are neighbors if they are joined by an edge, so the distance between them is 1. It is easy to see that if each computer $x$ uses the following algorithm for relaying a message with destination $y$, then messages are always routed along a minimal length path.

\begin{verbatim}
if $x = y$
  then accept message
else if $\delta(x, y) = \infty$
  then message undeliverable
else send message to a neighbor $p$ with the smallest value of $\delta(p, y)$
fi
fi
\end{verbatim}
In the network of Fig. 1, $B$ would send a message destined for $E$ to $C$ rather than to $A$ because $\delta(C, E) = 1$ and $\delta(A, E) = 2$. The optimal routing of messages is therefore easy if each computer has the following information:

- Its distance to every other computer;
- Who its neighbors are;
- The distance from each of its neighbors to every other computer.

For now, we assume that a computer knows who its neighbors are, and consider the problem of computing the distances.

Suppose $dist$ is any function satisfying the following relation for all vertices $b$ and $c$, where $1 + \infty$ is defined to equal $\infty$:

$$
\begin{align*}
dist(b, c) &= \text{if } b = c \\
& \quad \text{then } 0 \\
& \quad \text{else } 1 + \min\{dist(p, c) : p \text{ a neighbor of } b\}.
\end{align*}
$$

It is easy to show that this implies $dist(b, c) = \delta(b, c)$ for all $b$ and $c$.

This observation suggests the following iterative procedure for computing the distances, where assignment and equality of arrays is defined in the obvious way, and the for all statement is similar to a for statement, except that the different 'iterations' may be performed in any order – or concurrently.

```
repeat
    olddist := dist;
    for all computers $b$ and $c$
        do if $b = c$
            then $dist(b, c) := 0$
            else $dist(b, c) := 1 + \min\{olddist(p, c) : p \text{ a neighbor of } b\}$
        fi
    od
until $dist = olddist$.
```

Unfortunately, this program will not terminate for all initial values of $dist$. In particular, if $\delta(b, c) = \infty$ and initially $dist(b, c) < \infty$, then the value of $dist(b, c)$ will just keep increasing. To overcome this difficulty, we observe that if $NN$ is the number of vertices in the graph, then the distance between any two points is either less than $NN$ or else equals $\infty$. This means that if $dist$ satisfies the following relation for all $b$ and $c$:

$$
\begin{align*}
\text{if } b = c \\
\text{then } 0 \\
\text{else if } ndist(b, c) < NN - 1 \text{then } 1 + ndist(b, c) \\
\text{else } \infty
\end{align*}
$$

where $ndist(b, c) = \min\{dist(p, c) : p \text{ a neighbor of } b\}$, then $dist(b, c) = \delta(b, c)$ for all $b$ and $c$. 

(1)
This suggests the following algorithm for computing the distances:

\[ \text{repeat} \]
\[ \text{olddist} := \text{dist} ; \]
\[ \text{for all computers } b \text{ and } c \]
\[ \text{do if } b = c \]
\[ \text{then } \text{dist}(b, c) := 0 \]
\[ \text{else } \text{ndist}(b, c) := \min\{\text{olddist}(p, c) : \text{p a neighbor of } b\} \]
\[ \text{if } \text{ndist}(b, c) < NN - 1 \]
\[ \text{then } \text{dist}(b, c) := 1 + \text{ndist}(b, c) \]
\[ \text{else } \text{dist}(b, c) := \infty \]
\[ \text{fi} \]
\[ \text{fi} \]
\[ \text{od} \]
\[ \text{until } \text{dist} = \text{olddist}. \]

It is not hard to show that this program always terminates in at most \( NN \) iterations for any initial values of \( \text{dist} \), and that it terminates with \( \text{dist}(b, c) = \delta(b, c) \) for all \( b \) and \( c \). The proof involves observing that after the \( k \)th iteration, \( \text{dist}(b, c) = \delta(b, c) \) for all \( b \) and \( c \) such that \( \delta(b, c) < k \).

If the reader does not understand this algorithm, we advise him to apply it to the graph of Fig. 1, using arbitrary initial values, before proceeding.

We now transform this algorithm into a distributed one, in which each computer calculates its distance to every other computer – computer \( b \) calculating the array of values \( \text{dist}(b, -) \). Each value \( \text{olddist}(p, c) \) is used by every neighbor \( b \) of \( p \) in computing \( \text{dist}(b, c) \). In a distributed algorithm, these neighbors need their own copies of this value. We therefore introduce a new array \( \text{dtab} \) such that \( \text{dtab}(b, p, c) \) represents \( b \)'s copy of \( \text{olddist}(p, c) \). In place of the array assignment \( \text{olddist} := \text{dist} \), we let each computer \( b \) set each neighbor \( p \)'s subarray \( \text{dtab}(p, b, -) \) equal to its array \( \text{dist}(b, -) \). This yields the following algorithm:

\[ \text{repeat} \]
\[ \text{for all computers } b \]
\[ \text{do for all neighbors } p \text{ of } b \]
\[ \text{do } \text{dtab}(p, b, -) := \text{dist}(b, -) \text{ od} \]
\[ \text{od} \]
\[ \text{for all computers } b \]
\[ \text{do for all computers } c \]
\[ \text{do if } b = c \]
\[ \text{then } \text{dist}(b, c) := 0 \]
\[ \text{else } \text{ndist}(b, c) := \min\{\text{dtab}(b, p, c) : \text{p a neighbor of } b\} \]
\[ \text{if } \text{ndist}(b, c) < NN - 1 \]
\[ \text{then } \text{dist}(b, c) := 1 + \text{ndist}(b, c) \]
\[ \text{else } \text{dist}(b, c) := \infty \]
\[ \text{fi} \]
\[ \text{fi} \]
\[ \text{od} \]
\[ \text{od} \]
\[ \text{until dist}(b, c) = \text{dtab}(p, b, c) \text{ for all } b, c, p. \]

This can be regarded as a distributed algorithm, where every computer executes a separate iteration of each for all b statement. Moreover, computer b finishes with \( \text{dtab}(b, p, -) = \text{dist}(p, -) \) for every neighbor p, so the arrays dist and \( \text{dtab}(b, -, -) \) contain precisely the information b needs for routing messages. Unfortunately, this is a synchronous algorithm – every computer must finish executing its iteration of the first for all b statement before any computer can begin its iteration of the next one, and the termination condition is a global one that an individual computer cannot evaluate by itself. Such a synchronous algorithm is not well-suited for a network of independent computers.

In this synchronous algorithm, each computer b repeatedly cycles through the following two steps:

1. For every neighbor p: put each value dist(b, c) into p’s array element dtab(p, b, c),
2. Recompute all values dist(b, c), using dtab(b, -, -).

To turn it into an asynchronous algorithm, we make the following changes:
- Instead of putting the value dist(b, c) directly into dtab(p, b, c), b will send it to computer p, and p will put it into the dtab array;
- We reverse the order of the two steps. Since b repeatedly cycles through them, we can do this by simply executing the (original) first step before beginning the cycle. We then have a program in which b first sends all its values dist(b, -) to each of its neighbors, then cycles through the two steps:
  1. Compute the elements of dist(b, -);
  2. Send their values to its neighbors.
- If the value of dist(b, c) is not changed by the first step of this cycle, then there is no reason to send it in the second step. Thus, b will send its neighbors only the values of dist(b, -) that have changed.
- Computer b will execute this cycle whenever it receives a new value for dtab(b, p, c). Since a change in the distance from p to c cannot affect the distance from b to any computer other than c, b need recompute only the single element dist(b, c) when it receives this new value.

We then obtain an asynchronous algorithm in which each computer b does the following:

initialization: Send all the values dist(b, -) to each of its neighbors.
loop: Whenever a new value of dist(p, c) arrives from computer p:
  1. Set dtab(b, p, c) to this value;
  2. Recompute dist(b, c);
  3. If the value of dist(b, c) has changed, then send the new value to every neighbor q.
This algorithm works even when the sending and receipt of values is completely asynchronous. However, it is not our final algorithm. Although we have assumed a fixed network topology, our real goal is an algorithm for a network in which communication lines fail and are repaired. The distance between two computers is the shortest path along working communication lines, which changes when the set of working lines changes. We want an algorithm that recomputes these distances when they change.

Computing the distances requires that computers be able to determine who their current neighbors are. We assume that a computer is notified when any of its communication lines fails or is repaired. How this is done will not concern us. We require only that the following conditions be satisfied:

(1) Notifications of failures and repairs of any single communication line are received in the same order that the failures and repairs occur;

(2) Notification of a failure is received after the receipt of any message sent over the line before it failed;

(3) Notification of a repair is received before the receipt of any message sent over the line after it was repaired.

The failure of a line may result in the loss of messages that were in transit at the time of the failure, and of messages sent after the failure. Requirement (2) applies only to those messages that are not lost. A failure may cause the complete loss of a message, but we assume that it cannot garble a message. Any message that is delivered must be the same message that was sent.

We can modify our algorithm to handle failures and repairs by adding a new array $\text{nbrs}(b, -)$ to computer $b$, where $\text{nbrs}(b, c) = \text{true}$ if and only if $b$ thinks that $c$ is a neighbor. The value of $\text{nbrs}(b, p)$ can be set by $b$ when it receives a failure or repair notification for the line joining it with $p$. Computer $b$ handles the receipt of a new value of $\text{dist}(p, c)$ just as before, except using $\text{nbrs}(b, -)$ to determine who its neighbors are when recomputing $\text{dist}(b, c)$.

When $b$ is notified that the communication line joining it with $p$ has failed, it must set $\text{nbrs}(b, p)$ to $\text{false}$. Since the shortest path to any other node might have gone through $p$, computer $b$ must recompute its entire array $\text{dist}(b, -)$, and send its neighbors any values that change.

Other than setting $\text{nbrs}(b, p)$ to $\text{true}$, it is less obvious what $b$ should do when it learns that the line joining it with $p$ has been repaired. When the line is repaired, both $b$ and $p$ have no idea what values are in the other's $\text{dist}$ array. More precisely, the entries in $b$'s subarray $\text{dtab}(b, p, -)$ need not equal the corresponding entries in $p$'s array $\text{dist}(p, -)$, and vice-versa. Therefore, the first thing that $b$ and $p$ should do when they learn that the line has been repaired is to send each other the values of their $\text{dist}$ array elements.

Since $b$ does not know at this time what the entries in $\text{dtab}(b, p, -)$ should be, it should not use any of those entries. Recalling how $\text{dtab}$ is used, both for computing $\text{dist}(b, -)$ and routing messages, we see that an entry whose value is $\infty$ is essentially ignored. Therefore, when $b$ is first notified of the line's repair, all the
Proof of a distributed algorithm

181
elements of \( dtab(b, p, -) \) should be set to \( \infty \), with one exception: \( b \) knows that the
distance from \( p \) to itself is zero, so \( dtab(b, p, p) \) should equal 0 and \( dist(b, p) \) should
equal 1. Putting this all together, we see that \( b \) should perform the following actions
upon being notified that the line joining it with \( p \) has been repaired:

- Send all its values \( dist(b, -) \) to \( p \), except for \( dist(b, b) \), which \( p \) knows to be zero;
- Set \( nbrs(b, p) \) to \( true \);
- Set all elements of \( dtab(b, p, -) \) to \( \infty \) except for \( dtab(b, p, p) \), which is set to 0;
- Set \( dist(b, p) \) to 1.

There is one last trick that we will use. Instead of setting the elements of
\( dtab(b, p, -) \) to \( \infty \) when the line joining \( b \) and \( p \) is repaired, let them all, including
\( dtab(b, p, p) \), be set to \( \infty \) when the line fails. When the line is repaired, \( dtab(b, p, p) \)
is set to 0 and the other elements are left unchanged. This means that \( nbrs(b, p) \)
equals \( true \) when \( dtab(b, p, p) = 0 \), and it equals \( false \) when \( dtab(b, p, p) = \infty \). Hence,
the array \( nbrs \) is superfluous and can be eliminated. Moreover, since all the entries
of \( dtab(b, p, -) \) equal \( \infty \) when \( p \) is not a neighbor of \( b \), it is easy to see that the
minimum of \( \{dtab(b, p, c) : p \text{ a neighbor of } b\} \) is the same as the minimum of the
entire \( dtab(b, -, c) \) subarray, which simplifies the computation of \( dist(b, c) \).

Finally, we eliminate the need for a special initialization step by assuming that
initially all communication lines have failed and all entries of \( dist \) and \( dtab \) are \( \infty \),
except that \( dist(b, b) = 0 \) for all \( b \). When the system is started, it must generate the
appropriate repair notifications for the working communication lines.

Combining all these remarks, we obtain the following algorithm, where ‘recom-
puting’ \( dist(b, c) \) means setting it so it satisfies

\[
\begin{align*}
\text{dist}(b, c) &= \textbf{if } b = c \textbf{ then } 0 \\
&\quad \textbf{else if } \min\{\text{dtab}(b, -, c)\} < NN - 1 \\
&\quad \quad \textbf{then } 1 + \min\{\text{dtab}(b, -, c)\} \\
&\quad \quad \textbf{else } \infty,
\end{align*}
\]

and a ‘neighbor’ of \( b \) is any computer \( p \) such that \( dtab(b, p, p) = 0 \).

Algorithm for computer \( b \)

When a value for \( dist(p, c) \) is received from computer \( p \)
1. Set \( dtab(b, p, c) \) to this value.
2. Recompute \( dist(b, c) \).
3. If the value of \( dist(b, c) \) was changed, then send the new value to every
   neighbor of \( b \).

When notified of the failure of the line to computer \( p \)
1. Set all elements of \( dtab(b, p, -) \) to \( \infty \).
2. Recompute all elements of \( dist(b, -) \).
3. Send each value that was changed to every neighbor of \( b \).

When notified of the repair of the line to computer \( p \)
1. Set \( dtab(b, p, p) \) to 0 and \( dist(b, p) \) to 1.
2. Send all elements of $\text{dist}(b, -)$ to $p$ except for $\text{dist}(b, b)$.
3. Send $\text{dist}(b, p)$ to all neighbors of $b$.

This completes our informal description of the algorithm. Although differing in many details, it is basically the same as the algorithm given in [12], described with about the same degree of imprecision. There are two shortcomings of the algorithm that should be noted.
- The value of $\text{NN}$ – or at least an upper bound for its value – must be known in advance;
- A computer $b$ does not know when the algorithm has terminated and its distance table $\text{dist}(b, -)$ is correct. Other methods, such as the one in [2] must be used to discover this.

Before we can write a rigorous proof of its correctness, we will need a more precise statement of the algorithm. However, our informal description will suffice for the informal proof presented in the next section.

3. An informal correctness proof

We will prove the following property of the algorithm, which is essentially the same property proved in [12]:

If communication lines stop failing and being repaired, then the computers eventually obtain and maintain the correct values of $\text{dist}$ and $\text{dtab}$.

The values of $\text{dist}$ and $\text{dtab}$ are correct if:
- $\text{dist}$ satisfies Eq. (1) and
- for all $b$ and $p$:

  if there is a working communication line joining $b$ and $p$
  then $\text{dtab}(b, p, -) = \text{dist}(p, -)$
  else $\text{dtab}(b, p, -) = \infty$.

To prove this property, we prove the following two properties, where the system is said to be $\text{stable}$ if there are no unprocessed failure or repair notifications and no "$\text{dist}(b, c)$" messages in transit.

C1. If the system is stable, then the values of $\text{dist}$ and $\text{dtab}$ are correct.
C2. In the absence of failures and repairs, the system will eventually become stable.

These two properties are proved separately.

3.1. Proof of property C1

Property C1 is a safety property – one which asserts that some predicate $\mathcal{P}$ is always true. (A predicate is a boolean-valued function of the system state.) Such a safety property is proved by finding a predicate $\mathcal{I}$ with the following properties:
S1. \( I \) is true initially.
S2. Each action of the system leaves \( I \) true.
S3. \( I \) implies \( P \).

A predicate satisfying condition 2 is called an \textit{invariant} of the system. For a general discussion of this method, we refer the reader to [6] or [11].

In the proof of Cl, \( P \) is the assertion that if the system is stable then the values of \textit{dist} and \textit{dtab} are correct. Recalling (1) and (2) of Section 2, it is easy to see that this is equivalent to:

\[
P \equiv \begin{cases} 
\text{if the system is stable} \\
\text{then } P_1: \text{dist and } \text{dtab satisfy (2)} \quad \text{and} \\
\text{P}_2: \text{for all } b \text{ and } p: \\
\quad \text{if there is a working communication line joining } b \text{ and } p \\
\quad\text{then } \text{dtab}(b, p, -) = \text{dist}(p, -) \\
\quad\text{else } \text{dtab}(b, p, -) = \infty.
\end{cases}
\] (3)

This expression for \( P \) is convenient because the algorithm always recomputes the elements of \textit{dist} so as to keep condition \( P_1 \), true. To satisfy condition S3 – the only condition that mentions \( P \) – the invariant \( I \) must imply that \( P_1 \) and \( P_2 \) are satisfied when the system is stable.

The invariant \( I \) will be a conjunction \( I_1 \land I_2 \), where \( I_1 \) is defined in Eq. (3), and \( I_2 \) implies that \( P_2 \) holds when the system is stable. We want \( I_2 \) to assert that \( \text{dtab} \) will have the correct value when all outstanding \textit{"dist"} messages and all failure and repair notification have been processed. However, \( I_2 \) must be a predicate – a function of the current state of the system – so we have to translate the assertion \textit{“will have . . . when . . .”} into an assertion about the current state.

In order to formulate the predicate \( I_2 \), we must state more precisely our assumptions about the order in which a computer receives messages and notifications of failure and repair. For each pair of computers \( b \) and \( p \) that are joined by a communication line, we assume that there are three queues:

\[
\begin{align*}
OQ(p, b): & \quad \text{A queue of } \textit{"dist}(p, c)" \text{ messages in } p \text{'s output buffer waiting to be sent over the line;} \\
TQ(p, b): & \quad \text{A queue of } \textit{"dist}(p, c)" \text{ messages currently being transmitted over the line;} \\
IQ(p, b): & \quad \text{A queue of } \textit{"dist}(p, c)" \text{ messages and failure and repair notifications for the line that } b \text{ has received but not yet processed.}
\end{align*}
\]

Note that there are two sets of unidirectional queues for the bidirectional communication line joining \( b \) and \( p \): \( OQ(p, b), TQ(p, b) \) and \( IQ(p, b) \) for messages from \( p \) to \( b \); and \( OQ(b, p), TQ(b, p) \) and \( IQ(b, p) \) for messages from \( b \) to \( p \).
When the line is working, messages move from $OQ(p, b)$ to $TQ(p, b)$ to $IQ(p, b)$. When the line fails, a failure notification is placed at the end of $IQ(p, b)$ and the messages in $TQ(p, b)$ are thrown away. Thereafter, messages from $OQ(p, b)$ are thrown away instead of being moved into $TQ(p, b)$. When the line is repaired, a repair notification is placed at the end of $IQ(p, b)$ and normal message transmission is resumed. We let $Q(p, b)$ denote the concatenation of the queues $IQ(p, b)$, $TQ(p, b)$ and $OQ(p, b)$, so $Q(p, b)$ contains the entire queue of unprocessed messages that $p$ has sent to $b$, together with $b$’s unprocessed notifications of the failure and repair of the line joining $b$ and $p$.

We now derive a predicate which expresses the assertion that $drab$ will have the correct value when all outstanding messages and notifications have been processed. We first define a predicate $\mathcal{U}(b, p)$ which states that $dtab(b, p, p)$ will have the correct value when $b$ has finished processing all repair and failure notifications in $IQ(p, b)$.

$$\mathcal{U}(b, p) =$$

There are no "dist(p, p)" messages in $Q(p, b)$ and if the communication line joining $b$ and $p$ is working

then 1. if there is a failure or repair notification in $IQ(p, b)$

then A. the last such notification is a repair notification

else B. $dtab(b, p, p) = 0$

fi

else 2. if there is a failure or repair notification in $IQ(p, b)$

then A. the last notification in $IQ(p, b)$ is a failure notification

else B. $dtab(b, p, p) = \infty$

fi

fi

We next define a predicate $\mathcal{R}(b, p, c)$, for $c \neq b, p$, which essentially states that when $b$ and $p$ have finished processing their outstanding messages and notifications, then $dtab(b, p, c)$ will wind up with the correct value – i.e., with the current value of $dist(p, c)$ if the communication line joining them is working, and with $\infty$ if it isn’t. It is defined as follows:

$$\mathcal{R}(b, p, c) =$$

if the communication line joining $b$ and $p$ is working.

then 1. if there is no repair notification in $IQ(b, p)$

then if there is a "dist(p, c)" message in $Q(p, b)$

then A. the last such message must follow any repair or failure notifications, and must contain the current value of $dist(p, c)$.

else B. $dtab(b, p, c) = dist(p, c)$

and there are no failure or repair notifications in $IQ(p, b)$

fi
Proof of a distributed algorithm

else 2. if there is no failure notification in $IQ(p, b)$.
    then A. $dtab(b, p, c) = \infty$.
    fi
fi

The reader should be able to convince himself that $U(b, p)$ and $R(b, p, c)$ ought to remain true throughout the execution of the algorithm. We will show that they are, indeed, invariants of the algorithm.

We now define $I_2$ to be the conjunction of the predicates $U(b, p)$ and $R(b, p, c)$ for all $b, p$ and $c$ with $b \neq p$ and $c \neq b, p$. The invariant $I$ is $P_1 \land I_2$, where $P_1$ was defined in Eq. (3). To prove property $CI$, we must show that $I$ satisfies conditions S1-3.

We refer to the various conditions in the if...then...else structure of $U(b, p)$ and $R(b, p, c)$ by the indicated labels. For example, condition 2A of $R(b, p, c)$ states that the communication line joining $b$ and $p$ is not working, there is no failure notification in $IQ(p, b)$, and $dtab(b, p, c) = \infty$.

Condition S1 states that $I$ is true initially. Recall that the initial state is one in which all communication lines are failed, there are no unprocessed messages or notifications, and all values of $dist$ and $dtab$ equal $\infty$, except that $dist(b, b) = 0$ for all $b$. To prove S1, we must show that $P_1, U(b, p)$ and $R(b, p, c)$ are true in this state, for all $b, p$ and $c$. The reader can check that the truth of the predicates follows immediately from the initial assumptions.

We next verify condition S3, which states that $I$ implies $P$. By Eq. (3), this means we must show that if $I$ holds and the system is stable, then $P_1$ and $P_2$ hold. Since $I = P_1 \land I_2$, it suffices to show that if the system is stable, then the conjunction of all the $U(b, p)$ and $R(b, p, c)$ implies $P_2$. Recall that $P_2$ essentially states that the value of each $dtab(b, p, c)$ is correct – equaling $dist(p, c)$ if the communication line joining $b$ and $p$ is working, and $\infty$ if it isn't. In a stable state, condition 1B or 2B of $U(b, p)$ holds, which handles the case when $c = p$, and condition 1A or 2A of $R(b, p, c)$ holds, taking care of the case when $c \neq p$.

Finally, we must prove condition S2 – the invariance of $I$. This involves showing that if $I$ is true, then any action of the system leaves $I$ true. We have already observed that all system actions leave $P_1$ true, so we must show that they also leave $I_2$ true. We do this by showing separately that each $U(b, p)$ and $R(b, p, c)$ is invariant. From its definition, we see that the only actions which affect the truth value of any part of $U(b, p)$ are the following:
- The failure or repair of the communication line joining $b$ and $p$;
- Computer $b$ processing a failure or repair notification.

There are four actions, which are easily checked as follows:
- **Repair of the line**: This makes condition 1A true;
- **Failure of the line**: This makes condition 2A true;
- **Computer $b$ processes a repair notification**: The only case in which this could change the truth value of $U(b, p)$ is if the line is working and this is the last notification in $IQ(p, b)$. In this case, 1A must have been true before processing the notification, which ensures that 1B will be true afterwards.
- **Computer b processes a failure notification**: This cannot change the truth value of \( \mathcal{U}(b, p) \) unless the line is not working and \( b \) is processing the last notification in \( IQ(p, b) \). In this case, 2A must have been true before processing the notification, and 2B must become true afterwards.

Examining the definition of \( \mathcal{R}(b, p, c) \) shows that the following are the only actions that can affect the truth of any of its parts:
- The failure or repair of the communication line joining \( b \) and \( p \);
- Computer \( b \) processing a "dist\((p, c)\)" message or a failure notification for that line;
- Computer \( p \) processing a repair notification for the line joining it with \( b \).

There are five of these actions, which we consider separately:
- **Failure of the communication line joining \( b \) and \( p \)**: This places a failure notification at the end of \( IQ(p, b) \), making condition 2 vacuously true;
- **Repair of the communication line joining \( b \) and \( p \)**: This places a repair notification at the end of \( IQ(b, p) \), making condition 1 vacuously true;
- **Computer \( b \) processes a "dist\((p, c)\)" message**: This can change \( \mathcal{R}(b, p, c) \) only if 1A holds initially, there are no notifications in \( IQ(p, b) \), and this is the last "dist\((p, c)\)" message in \( Q(p, b) \). In this case, 1A implies that the message contains the current value of \( dist(p, c) \), so condition 1B will hold after \( b \) processes the message;
- **Computer \( b \) processes a failure notification from \( IQ(p, b) \)**: This could make \( \mathcal{R}(b, p, c) \) false only by making the if expression in condition 2 true. In this case, condition 2A will hold after \( b \) processes the notification;
- **Computer \( p \) processes a repair notification from \( IQ(b, p) \)**: This action can make the outer if condition true in condition 1 if it processes the last repair notification in \( IQ(b, p) \). However, the action causes \( p \) to send \( b \) a "dist\((p, c)\)" message, making condition 1A true in this case.

This completes the proof of the invariance of \( \mathcal{F} \), which finishes the proof of property \( C1 \).

### 3.2. Proof of property \( C2 \)

Property \( C2 \) is a *liveness* property – one which asserts that some predicate \( \mathcal{F} \) is eventually true. In our case, \( \mathcal{F} \) is the predicate that is true when the system is stable – i.e., when there are no unprocessed notifications or "dist\((p, c)\)" messages.

The traditional method of proving such a property is to find a nonnegative integer-valued function \( V \) of the system state such that if \( \mathcal{F} \) is not true, then the value of \( V \) will eventually decrease. Since a nonnegative integral function cannot keep decreasing forever, this implies that \( \mathcal{F} \) must eventually become true.

Instead of choosing \( V \) to have integer values, we can let its range be any set \( W \) with a well-founded total order relation \( > \), where an order relation is well-founded if there is no infinite decreasing chain

\[
w_1 > w_2 > \ldots \]
A common choice of $W$ is the set of all $n$-tuples of nonnegative integers, where $>$ is the lexicographical ordering defined for $n = 1$ to be the usual ordering, and defined inductively for $n > 1$ by $(a_1, \ldots, a_n) > (b_1, \ldots, b_n)$ if and only if $a_1 > b_1$, or else $a_1 = b_1$ and $(a_2, \ldots, a_n) > (b_2, \ldots, b_n)$.

In our proof, the function $V$ is defined to have as its value the following $NN + 1$-tuple (recall that $NN$ is the number of computers):

$$(e, m(1) + d(1), \ldots, m(NN - 1) + d(NN - 1), m(\infty) + d(\infty)),$$

where:

- $e =$ the total number of unprocessed repair and failure notifications,
- $m(i) =$ the number of unprocessed "dist" messages reporting values equal to $i$,
- $d(i) =$ the number of pairs $(b, c)$ such that $\text{dist}(b, c) = i$.

To prove C2, we show that in the absence of failures or repairs, if the system is not stable then the $NN + 1$-tuple $V$ must keep decreasing.

The value of $V$ can be affected only by the failure or repair of communication lines, and the processing of "dist" messages and failure or repair notifications. Since we are assuming that there are no further failures or repairs, we need only consider the latter three events. We show that each of them decreases $V$.

Since processing a failure or repair notification decreases the first (left-most) element of $V$, this obviously decreases its value. We therefore have only to consider the action of a computer $b$ processing a "dist$(p, c)$" message, which involves the following three steps:

1. Remove the message from the input queue $IQ(p, b)$.
2. Set $d_{ab}(b, p, c)$ equal to its value, and recompute $\text{dist}(b, c)$.
3. If the value of $\text{dist}(b, c)$ has changed, then send its new value to all the neighbors.

Let $i$ be the value that the message gives for $\text{dist}(p, c)$, let $j$ be the original value of $\text{dist}(b, c)$ and let $j'$ be its new value. These three actions affect the value of $V$ in the following ways:

1. Decreases $m(i)$ by one.
2. If $j' \neq j$, then decreases $d(j)$.
3. If $j' \neq j$, then increases $d(j')$ and $m(j')$.

We consider separately the following three cases:

- $j' = j$: In this case, the only effect is to decrease $m(i)$, which decreases $V$.
- $j' > j$: In this case, the component of $V$ that is increased – the $j'$-component – lies to the right of a component that is being decreased – namely, the $j$-component. Hence, $V$ is decreased.
- $j' < j$: In this case, processing the message decreases the value of $\text{dist}(b, c)$. This can only happen if the value $i$ that is received is smaller than any of the
other elements in $dtab(b, c)$, so $j' = i + 1$. Hence, it is only the $i + 1$-component that is increased while the $i$-component is decreased, so $V$ is decreased.

We thus showed that as long as there are no more failures or repairs, if there is an unprocessed failure or repair notification or "dist" message, then processing it will decrease the $NN + 1$-tuple $V$. This implies that in the absence of failures and repairs, the system must eventually reach a stable state, completing the proof of property C2, which completes our informal correctness proof of the algorithm.

4. Formalizing the proof

The informal proof in Section 3 is really a proof for a program in which the processing of a message or notification – removing it from the input queue, recomputing $dtab$ and $dist$, and putting the appropriate "dist(b, c)" messages on output queues – is an indivisible atomic operation. It is easy to write such a program in a concurrent programming language and translate our informal proof into a formal one for that program. However, a real implementation is not likely to use such large atomic operations; for example, it probably could send only one message with a single operation. We would then be left with the question of whether we had proved anything about a real implementation.

One approach is to prove the correctness of our coarse-grained program – the one with very large atomic operations – and then show that a real implementation is equivalent to it, where two programs are considered equivalent if for every execution of one there is an equivalent execution of the other. There are two problems with this approach:

- It assumes that we know what it means for two executions to be equivalent. Although there are formal definitions of this concept, they leave open the question of whether two 'equivalent' programs really are equivalent when used as components in a larger system.
- There are no well-developed formal methods for proving the equivalence of two very different realizations of an algorithm.

We therefore consider other approaches.

4.1. Medium-grained programs

The most obvious approach is to prove the correctness of the program that is actually being run on the computer. There are two reasons why one may not want to do this:

- Real programs have very small atomic operations – typically operations on single memory words – and assertional proofs of such fine-grained programs involve a great deal of uninteresting detail;
We may be interested in verifying the algorithm, not a particular implementation of it.

This suggests representing the algorithm with a medium-grained program – one whose atomic operations are small enough so we feel confident that it accurately represents a real implementation, yet large enough so the proof is not too complex. This is the approach we have taken.

4.1.1. Our program

In the Appendix, we sketch a formal correctness proof for an implementation of the algorithm as a medium-grained multiprocess program, with each computer represented by a node process and each communication line represented by an arc process. Communication between a node process and the adjacent arc processes is modeled by having the input and output queues be shared variables. The operations of inserting and removing an element from a shared queue are taken to be atomic. In our implementation, we have made the atomic operations as large as possible subject to the constraint that each atomic operation perform at most one access to a shared variable. It is well known that by observing this constraint, one can transform a program into an equivalent (in the sense described above) coarser-grained one [10].

As we remarked above, the proof of Section 3 may be regarded as the correctness proof of a coarse-grained program in which a node processes a single message or notification as one atomic operation. Our problem is to transform this proof into a correctness proof for the finer-grained program of the Appendix. In the coarse-grained program, a single atomic operation changes the value of several variables that are changed by separate operations in the finer-grained one. The finer-grained program has ‘intermediate’ states not present in the coarse-grained one – states in which the coarse-grained program’s operations are only partially completed. An invariant for the coarse-grained program need not be invariant for the finer-grained one because it need not hold in these intermediate states.

As an example, consider the invariant \( \mathcal{U}(b, p) \) of Section 3. Its invariance rests upon the assumption that the same atomic operation that effects the failure or repair of the communication line also puts the appropriate notifications in both its processes’ input queues. In our finer-grained program, a failure or repair involves the execution of three separate atomic operations: one that changes the status of the arc, and two subsequent operations that put notifications in the input queues. The truth of "the communication line joining \( b \) and \( p \) working" is changed by the first operation, which can make \( \mathcal{U}(b, p) \) become false until the two remaining operations put the notifications on the input queues.

Similarly, in the program given in the Appendix, a node processes a message or notification by recomputing \( dtab \) and \( dist \) with one atomic operation, then sending its \( "dist(b, c)" \) messages by separate operations. This means that \( \mathcal{R}(b, p, c) \) can be false when \( dtab \) and \( dist \) have been changed but the messages have not yet been sent.
Our proof does not work for the finer-grained program because that program does not simultaneously change several shared variables with a single operation the way the coarse-grained program does. To overcome this problem, we modify the coarse-grained program’s invariants by replacing references to program variables with state functions – where a state function is an arbitrary function of the value of variables and ‘program counters’. The state functions are chosen so that they are changed simultaneously by the fine-grained program, in the same way the coarse-grained program changes its shared variables.

For example, we replace $IQ(p, b)$ by a state function $IQ'(p, b)$ that equals $IQ(p, b)$ except when the arc process has just executed the atomic “fail” or “repair” operation that changes its internal state, but has not yet put the notification in $IQ(p, b)$. In that case $IQ'(p, b)$ equals the concatenation of $IQ(p, b)$ with that notification. The value of $IQ'(p, b)$ changes at the same time as the internal state of the arc, and $U(b, p)$ becomes an invariant of the finer-grained program when it is defined in terms of $IQ'(p, b)$ and $IQ'(b, p)$ instead of $IQ(p, b)$ and $IQ(b, p)$.

We use the same trick for $R(b, p, c)$. Its invariance for the coarse-grained program rests upon node b’s changing $dtab$ and $dist$, and putting messages on its output queues, all in a single atomic action. To achieve this in the finer-grained program, we define a state function $OQ'(b, p)$ that is the same as $OQ(b, p)$, except that it is changed by the same atomic operation that changes $dtab$ and $dist$ – being changed to the value that $OQ(b, p)$ will have when the node finishes sending its messages. We then let $Q'(p, b)$ be the concatenation of $IQ'(p, b)$, $TQ(p, b)$ and $OQ'(p, b)$, and define $R(b, p, c)$ in terms of $IQ'(b, p)$ and $Q'(p, b)$ to obtain an invariant for the finer-grained program.

The formal definitions of these ‘primed’ state functions are given in the Appendix. They require the use of predicates that explicitly mention a process’s control state – i.e., the value of its ‘program counter’. Some computer scientists have objected to the explicit use of control states in the assertions; they want assertions to mention only program variables. We do not understand these objections, since the control state is just as much part of a program’s state as the values of its variables. Indeed, every programmer knows that variables can often be eliminated by encoding their values in the control state, and that control structure can often be simplified by adding extra variables. Moreover, it is well-known that one cannot verify concurrent programs without reasoning about their control state [10].

To avoid mentioning the control state, one must introduce ‘dummy variables’ – program variables used only to encode the control state. The easiest way to do this for our program is probably to introduce $IQ'(b, p)$ and $OQ'(b, p)$ as dummy variables. We find it inelegant to add extra variables to the program just for the proof – especially, as in this case, when the values of these dummy variables can so easily be defined as functions of the real program state.

4.1.2. Other programming languages

It has become fashionable to eschew the use of shared variables and adopt programming languages in which interprocess communication is performed only
through value-passing communication primitives, as in CSP [3]. It is argued that this more restricted form of communication will make it easier to reason about programs, simplifying their proofs.

While there may be valid reasons for preferring these languages, simplifying correctness proofs is not one of them. The structure of a proof is determined by the underlying algorithm, not the language in which the algorithm is described. It is easy to express our algorithm in CSP, using the ‘!’ and ‘?’ operations to effect the communication between the nodes and the arcs. We can then just as easily translate our proof into a proof of the new program, essentially by changing the definitions of the state functions $IQ'(b, p)$ and $OQ'(b, p)$.

For the kind of reasoning sketched in the Appendix, the details of the programming language make little difference to the proof. We used shared variables because the formal proof methods for them are better-known than the corresponding ones for other communication primitives. The mechanization of a proof may be harder in some languages than in others, but this is due to lower-level issues like whether ‘aliasing’ of variable names can be detected at compile time, not to the interprocess communication mechanism.

4.1.3. Other medium-grained programs

In the Appendix, we tried to represent the algorithm with the most general program in which adding or removing a message or notification from a queue is an atomic operation. We believe that any other implementation with the same granularity can be regarded as a special case of ours, in the sense that any execution of it corresponds to a possible execution of ours. For example, a program in which each node has a single input queue, with each message and notification identifying its source, can be viewed as a particular implementation of our program in which each $IQ(p, b)$ is stored as a subqueue of the single input queue.

If a program is a special case of ours, then its proof will be a special case of our proof. More precisely, the correspondence between the two programs can be used to transform our proof into a proof of the other program. For example, for a program with a single input queue, references to the variable $IQ(p, b)$ will be replaced by the state function whose value is the subqueue of the input queue containing all messages and notifications from the arc joining $b$ with $p$.

4.2. Fine-grained programs

Despite the complexity of verifying a real implementation, with its many atomic operations, formal verification is the only way to guarantee the correctness of a concurrent program. Unverified programs tend to have subtle, timing-dependent errors that are unlikely to be discovered by testing.

If the program uses essentially the same algorithm, then our proof can be modified to prove its correctness. The modification involves the same technique used to go from the proof of the coarse-grained program in Section 3 to the proof of the medium-grained program in the Appendix – replacing variables by state functions.
However, the many small atomic operations of a real program, and the corresponding large number of different control states, make this a complex task, and the proof should be checked by computer to avoid errors. The construction of the proof is straightforward, but 'straightforward' does not necessarily mean 'easy'. In [8], we describe a method for simplifying this sort of proof transformation.

Of course, our proof can be transformed into a proof of the fine-grained program only if that program really implements the algorithm. Breaking the large atomic operations of our program into finer ones allows more interleaving of operations from the different processes, and this can introduce errors. Moreover, even if the fine-grained program is correct, its correctness may depend upon interprocess synchronization mechanisms that were not present in the coarser-grained program. For example, if adding or removing a message from a queue is not an atomic operation, then additional synchronization is needed to handle concurrent access to the queue by different processes. The proof must demonstrate the correctness of this synchronization, which requires additional reasoning not present in our proof.

4.3. Abstract specifications

Instead of verifying a particular program, one would probably prefer to prove the correctness of an abstract specification of the algorithm. Any particular program would then be verified by showing that it is a correct implementation of the specification. A general method for doing this is discussed in [7]. In this method, a specification has the form “there exist state functions $f_1, f_2, \ldots$ such that $\ldots$”, using temporal logic assertions to describe how the state functions $f_i$ change. To specify our algorithm, the queues and distance tables would be state functions instead of program variables, and we would write temporal logic assertions instead of a program to describe how they change. However, the specification and correctness proof for our algorithm would look very similar to the program and proof sketched in the Appendix.

Proving that a program meets such a specification requires defining the state functions of the specification as state functions of the program – i.e., as functions of variables and 'program counters'. This is essentially the same as replacing the variables of a coarser-grained program by state functions of a finer-grained one. The specification method can be viewed as a formal method for showing that a fine-grained description of an algorithm correctly implements a coarse-grained description. By writing temporal logic assertions instead of programs, one avoids the difficulties encountered in trying to prove the equivalence of two programs.

5. Conclusion

Finding an assertional proof requires transforming our ordinary reasoning about how the program behaves into assertional reasoning about why the program state
ensures the correct behavior. We have found that this can be difficult, but when it is done, the assertional proof almost always turns out to be easier to understand than the behavioral reasoning. Moreover, the assertional proof can be reduced to a series of formal logical steps, which is not true of ordinary behavioral proofs.

In this paper, we have presented one more example confirming our experience with assertional proofs. In Section 3 we gave an informal assertional proof that we feel is easier to understand, and at least as rigorous, as the behavioral proof in [12]. We then discussed various ways of formalizing the proof, one of which is developed in the Appendix.

There would be little point to a paper that simply confirms the virtues of ordinary assertional proofs, which by now should be well-known, except that the algorithm we have considered is a distributed one. The verification method we have used was developed for nondistributed programs communicating via shared variables, and there is a widespread belief that other methods are needed for distributed programs. This belief seems to rest upon three ideas:

- The method requires that the communication medium be represented by one or more processes. It is felt that there should be a more efficient method to describe interprocess communication.
- The method is based on the concept of a global state – a snapshot of the entire system at a single instant of time. We showed in [4] that there is no invariant method of defining a global state in a distributed system; any way of doing it involves an arbitrary choice. It is felt that a proof method should not be founded upon such a noninvariant concept.
- The method involves reasoning about the entire network of processes at once. It is felt that the distributed structure should be reflected in the proof method.

These objections originally convinced us that assertional proofs should not work for distributed algorithms. Our mind was changed only by attempting to prove the correctness of this algorithm. We now consider why these objections are not valid.

Representing the communication medium by separate processes turns out to be an advantage instead of a drawback, because it forces one to specify precisely the interprocess communication mechanism. In our algorithm, the precise specification of the communication lines, and how they fail and are repaired, is crucial. The description given in [12] is somewhat vague, and there is at least one ambiguity that could lead to an incorrect implementation.

Any proof method that simplifies the specification of interprocess communication – for example, by assuming message queues – at best provides only a set of predefined state functions for the user’s convenience, and at worst can limit the type of interprocess communication mechanism that it will handle. We doubt that there is any better way to specify the behavior of the communication lines in our algorithm than by describing each line as a process. (Of course, there are better ways to specify a process than by writing a program [7].)
The objection to the method because of its use of global states is a more profound one, and deserves a careful response—especially since we must assume some responsibility for it. One answer comes from an analogy with physics. Special relativity teaches that the way spacetime is split into space and time coordinates depends upon the observer, and is not invariant. Yet, one must choose space and time coordinates to perform numerical calculations. This causes no problem, since the results of a calculation do not depend upon the choice of coordinates. Similarly, the validity of an assertional correctness proof does not depend upon how one defines the global state, so one is free to choose an arbitrary definition.

This argument can be made more precise. Choosing a global state is the same as choosing a particular way of interleaving the atomic actions of the different processes into a single execution sequence. For a temporal logic assertion to be provable, it must be true for all possible program executions, and in particular, it must be true for all the possible interleavings obtained from any real execution. Our proofs are invariant because they involve an implicit quantification over all ways of defining the global state.

The final objection, that the proof method does not reflect the distributed structure, is one that has been advanced in support of assertional methods especially designed for distributed systems. It results from a confusion between distribution and modularity. The benefits of modularity are well established, and it is felt that the decomposition of a distributed system into separate processes should also provide a decomposition into separate modules. However, modularity should reflect the decomposition into logically separate areas of concern. Components that logically are intimately related should be part of the same module, even if they are physically remote from one another.

We have considered one module in a message-passing system—the module that maintains the routing tables. This module is implemented with one process at each site. There may be other modules that are implemented at a single site—e.g., an archive facility implemented with a very large data storage device on one of the computers. There are advantages to implementing a single system function at a single site, so the modular decomposition of the system will tend to reflect its physical distribution. However, a distributed system will also contain modules that are implemented at several sites, and are responsible for maintaining some form of consistency among those sites.

A distributed algorithm is not designed by specifying each process separately, and hoping they work properly when they are plugged together. The design of a distributed algorithm requires reasoning about all the processes together, and it is this global reasoning that is reflected in assertions about the global state.

These arguments for and against the use of global reasoning are philosophical ones. When philosophy and observation conflict, it is philosophy that must be revised. Our strongest argument for the use of global assertional reasoning about distributed programs is the example that we have presented.
Appendix. A formal proof

We now sketch a formal proof for a particular realization of our algorithm as a multiprocess program. We first describe the program, then give the proof.

A.1. The program

The execution of a multiprocess program is formally specified as an interleaving of the atomic actions of the individual processes. The state of the program, consisting of the values of all variables and of the ‘program counters’ of all processes, is defined before and after each atomic action.

An atomic operation is a portion of a process’s program whose execution is an atomic action. When writing a multiprocess program, one must indicate what the atomic operations are. We do this by enclosing atomic operations in angle brackets. Thus, execution of the statement

\[
\text{if } (b) \text{ then } (S) \text{ else } (T)
\]

consists of two atomic actions:
1. evaluation of the boolean condition \( b \),
2. execution of either \( S \) or \( T \).

We will not bother to formally define a programming language; instead, we use simple language constructs that are easy to understand. However, we do need a few unusual language features that require an explanation.
- We let ‘?’ denote a boolean condition that nondeterministically has the value \( \text{true} \) or \( \text{false} \). Thus,

\[
\text{if } (?) \text{ then } S \text{ else } T
\]

is a statement that nondeterministically chooses to execute either \( S \) or \( T \).
- We use an \( \text{iff} \ldots \text{fi} \) statement that is the same as Dijkstra’s \( \text{if} \ldots \text{fi} \) described in [1], except that if none of the guards is true then the statement is equivalent to a \( \text{skip} \). (In Dijkstra’s command, an \( \text{if} \) with all guards false is an \( \text{abort} \).) Note that there is nondeterminism when more than one guard is true. We use it with angle brackets around each guarded command, indicating that the evaluation of the guard and, if it is true, the execution of the following statement list, is a single atomic action.
- The statement

\[
\text{cobegin } S_1 \sqcup \ldots \sqcup S_n \text{ coend}
\]

indicates that the statements \( S_i \) are to be executed concurrently. Note that if the \( S_i \) are atomic operations, then it simply executes these \( n \) operations in any order.
- The statement

\[
\text{for all } p \text{ do } S \text{ od}
\]
is an abbreviation for

\[ \text{cobegin } S_1 \; \ldots \; \ldots \; S_{NN} \; \text{coend}, \]

where \( S_i \) is the statement obtained by substituting \( i \) for \( p \) in \( S \). This is generalized in the obvious way to obtain statements such as "\text{for all } p \neq b\" and "\text{for all } p, q\".

- The statement \( x : \in X \) sets \( x \) equal to an arbitrary element of the set \( X \). It is undefined if \( X \) is empty. (The operator \( : \in \) is the analogue of \( := \) with \( : \in \) replacing \( := \).)

Our program will use the following object types:

- **distance**: a nonnegative integer or \( \infty \).
- **node**: an integer from 1 through \( NN \).
- **message**: an ordered pair \((\text{distance}(\text{message}), \text{node}(\text{message}))\), where \( \text{distance}(\text{message}) \) is of type \( \text{distance} \) and \( \text{node}(\text{message}) \) is of type \( \text{node} \). It represents a message stating that the distance from the sender to \( \text{node}(\text{message}) \) is \( \text{distance}(\text{message}) \).
- **notification**: a type consisting of the two values "fail" and "repair".
- **queue**: a sequence of elements. The following functions are defined on queues, where \( \phi \) denotes the empty queue:
  - \( \text{head}(q) \): The first element in the queue \( q \) — undefined if \( q = \phi \);
  - \( \text{tail}(q) \): The sequence of all elements in \( q \) except the first, also undefined if \( q = \phi \);
  - \( q \circ q' \): The concatenation of queues \( q \) and \( q' \). If \( q \neq \phi \), then \( \text{head}(q \circ q') = \text{head}(q) \). For a single element \( m \), \( q \circ m \) is defined in the obvious way to be the queue obtained by adding \( m \) to the end of \( q \).

We represent the algorithm by a program containing a process for each computer and for each communication line. We let the process \( \text{node}(b) \) represent computer \( b \), and the process \( \text{arc}\{b, p\} \) represent the communication line joining \( b \) and \( p \). Note that \( \{b, p\} \) is an unordered pair, so \( \text{arc}\{b, p\} \) and \( \text{arc}\{p, b\} \) are the same process. For simplicity, we assume such an arc process for every pair of distinct nodes; nonexistent communication lines are represented by arcs that are never repaired. Thus, our program is of the form

\[ \text{cobegin} \]
\[ \text{for all } b \text{ do } \text{node}(b) \text{ od} \]
\[ \square \]
\[ \text{for all } \{b, p\} \text{ do } \text{arc}\{b, p\} \text{ od} \]
\[ \text{coend}. \]

Interprocess communication is performed using the following shared variables:

- **OQ**(\( p, b \)): A queue of elements of type \( \text{message} \), accessed by \( \text{node}(p) \) and \( \text{arc}\{b, p\} \). It is initially empty.
$IQ(p, b)$: A queue of elements of type message or notification, accessed by $arc\{b, p\}$ and $node(b)$. It is initially empty.

These are the same queues that were described in our informal proof. The queue $TQ(p, b)$ is a variable local to $arc\{b, p\}$. We assume that the following two operations are atomic:
- Adding an element to a queue;
- Testing if a queue is empty, and removing its head if it is not.

We first give the program for $arc\{b, p\}$. It has the following local variables:

- $up\{b, p\}$: A boolean, which will be true if and only if the communication line joining $b$ and $p$ is working. It is initially false.
- $TQ(b, p)$: A queue of elements of type message, initially empty.
- $TQ(p, b)$: A queue of elements of type message, initially empty.

The arc process must model the following communication line actions:
- Failing and being repaired;
- Moving messages between the computers when the line is working;
- Throwing away messages in the output queues when the line is not working.

It is given by the following program:

\[
\text{PROCESS } arc\{b, p\} := \\
\text{loop forever} \\
\text{beg}(b, p); \text{if } (up\{b, p\}) \\
\text{then if } (?) \\
\text{then } fl\{b, p\}: \text{fail}\{b, p\} \\
\text{else } \text{move.message}\{b, p\} \\
\text{fi} \\
\text{else if } (?) \\
\text{then } rp\{b, p\}: \text{repair}\{b, p\} \\
\text{else } \text{throwaway.message}\{b, p\} \\
\text{fi} \\
\text{fi} \\
\text{end loop.}
\]

To complete the specification of $arc\{b, p\}$, we have to specify the $fail$, $repair$, $move.message$ and $throwaway.message$ operations. The $fail$ and $repair$ operations are straightforward, and are given below. Note the use of the $cobegin$ to avoid specifying whether the notification appears first on the input queue of $b$ or $p$, maintaining the symmetry in $b$ and $p$.

\[
\text{fail}\{b, p\} := \\
\text{begin} \\
fl\{b, p\}: \langle TQ(b, p) := TQ(p, b) := \phi; \\
up\{b, p\} := \text{false} \rangle;
\]


The move.message operation tries to move a message along the transmission line – either moving it from the output queue $OQ$ to the transmission queue $TQ$, or from $TQ$ to the input queue $IQ$. The throwaway.message operation deletes a message from an output queue. Note the use of the iff construction to make a nondeterministic choice of what to do.

move.message$\{b, p\}$ :=

iff
\[
\begin{align*}
(OQ(b, p) \neq \phi & \rightarrow TQ(b, p) := TQ(b, p) \circ head(OQ(b, p)); \\
& OQ(b, p) := tail(OQ(b, p))
\end{align*}
\]

iff
\[
\begin{align*}
(TQ(b, p) \neq \phi & \rightarrow IQ(b, p) := IQ(b, p) \circ head(TQ(b, p)); \\
& TQ(b, p) := tail(TQ(b, p))
\end{align*}
\]

iff
\[
\begin{align*}
(OQ(p, b) \neq \phi & \rightarrow TQ(p, b) := TQ(p, b) \circ head(OQ(p, b)); \\
& OQ(p, b) := tail(OQ(p, b))
\end{align*}
\]

iff
\[
\begin{align*}
(TQ(p, b) \neq \phi & \rightarrow IQ(p, b) := IQ(p, b) \circ head(TQ(p, b)); \\
& TQ(p, b) := tail(TQ(p, b))
\end{align*}
\]

iff

throwaway.message$\{b, p\}$ :=

iff
\[
\begin{align*}
(OQ(b, p) \neq \phi & \rightarrow OQ(b, p) := tail(OQ(b, p)))
\end{align*}
\]

iff
\[
\begin{align*}
(OQ(p, b) \neq \phi & \rightarrow OQ(p, b) := tail(OQ(p, b)))
\end{align*}
\]

Next, we give the program for node$(b)$. It consists of repeatedly performing the following three steps:
1. Find out which input queues are nonempty.
2. Choose a nonempty queue, and:
   - Find and delete the first element of that queue.
   - Update $d_{ab}$ and $\text{dist}$ appropriately.
   - Decide what "$\text{dist}(b, c)$" messages to send.
3. Send the "$\text{dist}(b, c)$" messages.

Since the second step accesses only a single shared variable (the chosen input queue), it can be made an atomic operation.

The program for process $\text{node}(b)$ uses the following local variables:

- $S(b)$: a set of nodes. It is the set of all nonempty input queues - i.e., $q \in S(b)$ if $b$ finds $\text{IQ}(q, b)$ nonempty in step 1;
- $p$: a node. It denotes the input queue that $b$ chooses to process in step 2 - i.e., $b$ will process the first message or notification in $\text{IQ}(p, b)$;
- $q, c$: nodes. They are used in for all statements;
- $\text{send}(b, -, -)$: a boolean array, each element of which initially equals false. The element $\text{send}(b, p, c)$ will equal true when $b$ intends to send a message to $p$ with the current value of $\text{dist}(b, c)$;
- $\text{next}$: of type message or notification. It is used to hold the message or notification currently being processed.

Note that unlike $S$ and $\text{send}$, we do not subscript the variables $p, q, c$ and $\text{next}$, using the same variable names for all nodes $b$. We do not need to distinguish between different instances of these variables because they are not mentioned in the proof.

The program for $\text{node}(b)$ is given below in terms of the operations $\text{process.message}$, $\text{process.failure}$ and $\text{process.repair}$, which are described afterwards. These operations in turn are specified in terms of a $\text{recompute.dist}(b, c)$ operation, which recomputes the value of $\text{dist}(b, c)$ and sets the appropriate elements of $\text{send}(b, -, c)$ to cause that value to be sent to all neighbors if it has changed.

\begin{verbatim}
PROCESS node(b) ::= 
  begin loop
    ck(b): forall q ≠ b
      do (if $\text{IQ}(q, b) \neq \phi$
          then $S(b) := S(b) \cup \{q\}$)
      od
    pr(b): (if $S(b) \neq \phi$
      then $p \in S(b)$;
        $\text{next} := \text{head}(\text{IQ}(p, b))$;
        $\text{IQ}(p, b) := \text{tail}(\text{IQ}(p, b))$;
  od
\end{verbatim}
if type(next) = message
    next = "fail"
    → process.failure
fi

next = "repair"  → process.repair
fi

for all q, c
    do
        if send(b, q, c)
            then
                OQ(b, q) := OQ(b, q) ∪ (dist(b, c), c);
                send(b, q, c) := false
            fi
        fi
    od
end loop

process.message ::= begin
    dtab(b, p, node(next)) := distance(next);
    recompute.dist(b, node(next))
end

process.failure ::= begin
    for all c
        do	dtab(b, p, c) := ∞;
            recompute.dist(b, c)
        od
end

process.repair ::= begin
    dtab(b, p, p) := 0;
    dist(b, p) := 1
    for all c ≠ b
        do send(b, p, c) := true;
            if dtab(b, c, c) = 0 ∧ c ≠ p
                then send(b, c, p) := true
            fi
        od
end.

To complete our program, we need only specify the operation recompute.dist(b, c). This operation is specified below, using the local variable newdist(b, c) which is of type distance.

recompute.dist(b, c) ::= newdist(b, c) := if b = c then 0
else if min{dtab(b, c)} < NN - 1
    then 1 + min{dtab(b, c)}
else ∞
Proof of a distributed algorithm

if \(\text{newdist}(b, c) \neq \text{dist}(b, c)\)
then \(\text{dist}(b, c) := \text{newdist}(b, c)\);
for all \(q \neq b\)
    do if \(\text{dtub}(b, q, q) = 0\)
        then \(\text{send}(b, q, c) := \text{true}\) fi
    od fi

A.2 The proof

In Section 4, we discussed the general approach to proving the correctness of our program. We have to define state functions \(IQ'(p, b)\) and \(OQ'(p, b)\) that change at the right time to maintain the invariance of \(\mathcal{U}(b, p)\) and \(\mathcal{R}(b, p, c)\). To do this, we first define the following functions of a queue \(q\):

\[
[q : b(\cdot)]: \text{the subqueue of } q \text{ consisting of all elements } x \text{ such that } b(x) = \text{true}, \text{ where } b \text{ is a boolean function. For example,}
\]
\[
[IQ'(b, p) : \text{type}(\cdot) = \text{notification}]
\]
is the queue consisting of all the notifications in \(IQ'(b, p)\);

\(\text{last}(q)\): the last element in \(q\). It is undefined if \(q = \emptyset\).

To define the 'primed' queues, we must refer to the value of a process's 'program counter'. This is done with the following boolean functions, defined for any labeled program statement \(s : S\).

\(\text{at } s\): true if control is at the beginning of statement \(S\);
\(\text{in } s\): true if control is at the beginning or anywhere inside statement \(S\), but not at an exit point of \(S\);
\(\text{after } s\): true if control is at the exit point of \(S\).

Note that if \(S\) is an atomic operation, then \(\text{at } s\) and \(\text{in } s\) are synonymous.

These predicates are defined more precisely in [5] and (11). We just note here that in the \(\text{fail}\) operation, \(\text{at fl}_2(b, p)\) and \(\text{at fl}_2(p, b)\) become true immediately after execution of the atomic statement labeled \(\text{fl}_1\{b, p\}\). A similar remark applies to the \(\text{repair}\) operation as well. There is no control point at the beginning of a \(\text{cobegin}\) before the start of its subprocesses.

We can now formally define the state functions \(IQ'(p, b), OQ'(p, b)\) and \(Q'(p, b)\) as follows:

\[
IQ'(p, b) = \begin{cases} 
\text{if } \text{at fl}_2(p, b) \\
\text{then } IQ(p, b) \odot \text{"fail"} \\
\text{else if } \text{at rp}_2(p, b) \\
\text{then } IQ(p, b) \odot \text{"repair"} \\
\text{else } IQ(p, b) 
\end{cases}
\]
\[ \mathcal{OQ}'(p, b) = \mathcal{OQ}(p, b) \circ \text{tosend}(b, p, 1) \circ \cdots \circ \text{tosend}(b, p, NN) \]

where \( \text{tosend}(b, p, c) = \text{if } \text{send}(b, p, c) \text{ then } (\text{dist}(b, c), c) \) 

\[ \text{else } \phi \]

\[ \mathcal{Q}'(p, b) = \mathcal{IQ}'(p, b) \circ \mathcal{TQ}(p, b) \circ \mathcal{OQ}'(p, b). \]

We now precisely define all the predicates introduced informally in Section 3.

\[ \mathcal{P}_1 = \forall b, c: \text{dist}(b, c) = \text{if } b = c \]

\[ \text{then } 0 \]

\[ \text{else if } \min\{\text{dtab}(b, -, c)\} < NN - 1 \]

\[ \text{then } 1 + \min\{\text{dtab}(b, -, c)\} \]

\[ \text{else } \infty \]

\[ \mathcal{P}_2 = \forall b: \forall p \neq b: \text{if up\{b, p\} = true} \]

\[ \text{then } \forall c: \text{dtab}(b, p, c) = \text{dist}(p, c) \]

\[ \text{else } \forall c: \text{dtab}(b, p, c) = \infty \]

\[ \text{stable } = \forall b: \forall p \neq b: \mathcal{Q}'(p, b) = \phi \land (\neg \in \text{fl\{b, p\}}) \land (\neg \in \text{rp\{b, p\}}) \]

\[ \mathcal{P} = \text{if stable then } \mathcal{P}_1 \land \mathcal{P}_2 \]

\[ \mathcal{U}(b, p, c) = \]

\[ \text{if up\{b, p\} = true} \]

\[ \text{then 1. if } [\mathcal{IQ}'(p, b): \text{type}(\cdot) = \text{message}] \neq \phi \]

\[ \text{then A. } \text{last}([\mathcal{IQ}'(p, b): \text{type}(\cdot) = \text{message}]) = \text{"repair"} \]

\[ \text{else B. } \text{dtab}(b, p, p) = 0 \quad \text{fi fi} \]

\[ \text{else 2. if } [\mathcal{IQ}'(p, b): \text{type}(\cdot) = \text{notification}] \neq \phi \]

\[ \text{then A. } \text{last}([\mathcal{IQ}'(p, b): \text{type}(\cdot) = \text{notification}]) = \text{"failure"} \]

\[ \text{else B. } \text{dtab}(b, p, p) = \infty. \quad \text{fi fi} \]

\[ \mathcal{R}(b, p, c) = \]

\[ \text{if up\{b, p\} = true} \]

\[ \text{then 1. if } [\mathcal{IQ}'(p, b): \cdot = \text{"repair"}] = \phi \]

\[ \text{then} \]

\[ \text{if } [\mathcal{Q}'(p, b): \text{type}(\cdot) = \text{message} \land \text{node}(\cdot) = c] \neq \phi \]

\[ \text{then A. } \text{type} (\text{last}([\mathcal{Q}'(p, b): \text{type}(\cdot) = \text{message} \land \text{node}(\cdot) = c] \lor (\text{type}(\cdot) = \text{notification}])) \]

\[ \land \]

\[ \text{distance} (\text{last}([\mathcal{Q}'(p, b): \text{type}(\cdot) = \text{message} \land \text{node}(\cdot) = c])) \]

\[ = \text{dist}(p, c) \]

\[ \text{else B. } \text{dtab}(b, p, c) = \text{dist}(p, c) \land \]

\[ [\mathcal{IQ}'(p, b): \text{type}(\cdot) = \text{notification}] = \phi \quad \text{fi} \]

\[ \text{fi} \]
else 2. if \([IQ'(p, b) \cdot = \text{"failure"}] = \emptyset\)

\[\text{then } \Delta. \ dlab(b, p, c) = \infty. \fi\]

There is one invariant needed for the proof of the finer-grained program that was not needed for the coarser-grained one. Recall that \(S(b)\) holds the names of the input queues that node \(b\) finds to be nonempty. We must state the obvious assertion that when control is at \(pr(b)\), after \(b\) has computed \(S(b)\) but before it has used it, these queues are still nonempty. We therefore define the following predicate:

\[\forall(b) = \text{if at } pr(b) \text{ then } \forall p \in S(b) : IQ(p, b) \neq \phi.\]

We then have the following invariant:

\[I = \forall b : \forall(b) \land \forall p \neq b: \mathcal{U}(b, p) \land \forall c \neq b, p : A(b, p, c).\]

We also specify the initial condition as follows:

\[\text{init } = \forall b : \text{at } \text{ck}(b) \land \text{dist}(b, b) = 0 \land \forall p \neq b: IQ(b, p) = IQ(b, p) = OQ(b, p) = \emptyset \land \text{dist}(b, p) = \infty \land \text{at } \text{beg}[b, p] \land \text{up}[b, p] = false \land \forall c \neq b : \text{dlab}(b, p, c) = \infty.\]

Finally, we define the following predicate, to be used in stating the correctness condition, which asserts that no arc is in the process of failing or being repaired:

\[\forall(b) \equiv \forall p \neq b : \neg((\text{fl}[b, p] \lor \text{rp}[b, p])).\]

The correctness condition for our algorithm is stated formally using temporal logic. A temporal logic formula may be thought of informally as an assertion about some specific time during the program execution. A predicate is a temporal assertion about the state at the current time. More general temporal assertions are formed with the unary operators \(\Box\) and \(\Diamond\), which have the following interpretations for any assertion \(A\):

\(\Box A\): the assertion \(A\) is true now and at all times in the future;

\(\Diamond A\): the assertion \(A\) is true now or at some time in the future.

A formal definition of this temporal logic can be found in [6] or [11]. We note here that \(\Box A = \neg \Diamond \neg A\) for any assertion \(A\). (We are thus using the ‘linear time’ logic of [6].)

Our correctness condition, that all computers eventually obtain the correct values of \(\text{dist}\) if lines stop failing and being repaired, is expressed formally by the assertion

\[\text{init } \Rightarrow \Box((\Box \text{no.arc.change}) \Rightarrow \Diamond \Box(\mathcal{P}_1 \land \mathcal{P}_2)). \tag{4}\]

The formal proof of this assertion follows the lines of our informal proof. First, we restate correctness conditions \(C1\) and \(C2\) formally as follows:

\[C1 = \text{init } \Rightarrow \Box \mathcal{P}\]

\[C2 = \text{init } \Rightarrow \Box(\Box \text{no.arc.change} \Rightarrow \Diamond \Box \text{stable}).\]
It is a simple exercise in temporal logic to show that \( C_1 \) and \( C_2 \) imply (4).

To verify \( C_1 \), we prove the following three assertions:

S1. \( \text{init} \to \mathcal{I} \)
S2. \( \mathcal{I} \to \square \mathcal{I} \)
S3. \( \mathcal{I} \to \mathcal{P} \).

Verifying S1 and S3 are simple exercises in logical deduction. The heart of the proof involves verifying S2, which asserts the invariance of \( \mathcal{I} \). We now sketch the general approach to proving such an invariance property.

For any program \( \Pi \) and any predicate \( \mathcal{I} \), we define \( \vdash \{ \mathcal{I} \} \Pi \{ \mathcal{I} \} \) to mean the following:

Executing any single atomic operation of program \( \Pi \) starting from a state in which \( \mathcal{I} \) is true results in a state in which \( \mathcal{I} \) is true.

In [5], we present a method for deriving such properties of a concurrent program \( \Pi \) from similar properties of the components of \( \Pi \). It is a generalization of the standard Hoare method for sequential programs, and can be viewed as a reformulation and generalization of the 'Gries–Owicki' method for concurrent programs [9]. (The Hoare method corresponds to the case where \( \Pi \) is a single atomic operation.) To go from such a generalized Hoare logic assertion to a temporal logic assertion, we need the following inference rule:

\[
\text{if } \vdash \{ \mathcal{I} \} \Pi \{ \mathcal{I} \} \text{ holds, where } \Pi \text{ is the entire program}
\]
\[
them \mathcal{I} \to \square \mathcal{I}.
\]

To prove S2, we must prove \( \vdash \{ \mathcal{I} \} \Pi \{ \mathcal{I} \} \), where \( \Pi \) is our multiprocess program. This involves proving the following for each atomic operation \( s: \langle S \rangle \) of \( \Pi \):

\[
\vdash \{ \text{at } s \land \mathcal{I} \} \{ \text{after } s \land \mathcal{I} \}.
\]

Except for the presence of "at, in and after" predicates, this is an ordinary Hoare-style assertion, and is proved by the same methods as for sequential programs.

The formal proof of the invariance of \( \mathcal{I} \) involves verifying this Hoare condition for every atomic operation of each process in our program. It is a formalization of the informal proof of Section 3, and will be omitted. The only new feature is the fact that while node \( b \)'s operation of inserting a message into \( OQ(b, p) \) does not add any new elements to \( OQ'(b, p) \), it can change the order of the elements. However, it cannot move a "\( \text{dist}(b, c) \)" message in front of a similar message for the same \( c \), and obviously cannot move it in front of a notification, so the operation leaves \( \mathcal{R}(b, p, c) \) unchanged.

Condition \( C_2 \) is a liveness property. Proving such a property usually requires making some assumption about fairness - that the interleaving of the operations from different processes is performed fairly, with no process ever omitted forever. For our algorithm, we must assume that the scheduling of the node and arc processes is fair. (This is a perfectly natural assumption, since we are interested in the case
in which they are implemented by physically distinct computers and communication lines.)

The question of fairness arises whenever there is nondeterminism. Each of our processes makes seemingly important nondeterministic choices - in \textit{node}(b), the choice of element from \textit{S}(b); in \textit{arc}\{b, p\}, the choice of which queue to take a message from. The correctness of the program does not require any fairness in these choices. Thus, \textit{node}(b) is free to ignore an individual input queue so long as there is any other nonempty input queue; \textit{arc}\{b, p\} can choose to ignore messages sent from \textit{b} to \textit{p} so long as there are messages going from \textit{p} to \textit{b}.

Having proved \textit{C1}, to prove \textit{C2} it suffices to prove

\[
\Box (\mathcal{F} \land \text{no.arc.change}) \Rightarrow \Box \Box \text{stable}. \tag{5}
\]

It is easy to prove the safety property

\[
\Box (\mathcal{F} \land \text{no.arc.change} \land \text{stable}) \Rightarrow \Box \text{stable},
\]

which states that in the absence of failures and repairs, once the system reaches a stable state, it will remain in it. Hence, to prove assertion (5), it suffices to prove

\[
\Box (\mathcal{F} \land \text{no.arc.change}) \Rightarrow \Box \text{stable}, \tag{6}
\]

where the temporal relation \(\Rightarrow\), pronounced "leads to"., is defined by

\[
\mathcal{F} \Rightarrow \mathcal{G} = \Box (\mathcal{F} \Rightarrow \Box \mathcal{G}).
\]

The "counting down" argument of our informal proof is formalized as follows.

We find a state function \(V\) having values in a set with a well-founded order relation \(<\) such that for any constant \(V_0\):

\[
[\Box (\mathcal{F} \land \text{no.arc.change}) \land V - V_0] \Rightarrow [\text{stable} \lor V < V_0]. \tag{7}
\]

Temporal logic reasoning shows that (7) implies (6).

The state function \(V\) is the \(NN + 1\)-tuple

\[
(e, m(1) + d(1), \ldots, m(NN - 1) + d(NN - 1), m(\infty) + d(\infty)),
\]

with \(e, m(i)\) and \(d(i)\) defined formally as follows, where \(|X|\) denotes the number of elements in the set or sequence \(X\):

\[
e = \sum_{b,p} [[IQ'(b, p): \text{type}(\cdot) = \text{notification}]],
\]

\[
m(i) = \sum_{b,p} [[Q'(b, p): \text{type}(\cdot) = \text{message} \land \text{distance}(\cdot) = i]],
\]

\[
d(i) = |\{(b, c): \text{dist}(b, c) = i\}|.
\]

Assertion (7) is proved by the method sketched in Section 3. The method for translating from that informal proof into a formal temporal logic proof is described in [11].
References