A Semiformal Correctness Proof Of A Network Broadcast Algorithm

Devendra Kumar

Department of Computer Sciences
The City College
and The Graduate Center
of The City University of New York
New York, NY 10031

Sitharam S. Iyengar

Computer Science Department
Louisiana State University
Baton Rouge, Louisiana 70803-4020

Abstract

In past years, a large number of published distributed algorithms have been shown to be incorrect. Unfortunately, designers of distributed algorithms typically use informal *correctness proofs, which tend to be unreliable.* Formal *correctness proofs offer a much higher degree of reliability, but they are not popular among algorithm designers because they are too mathematical and they typically assume synchronous message communication or some other abstract notation, and are therefore not easily applicable to the asynchronous message passing environment — the environment commonly assumed by many algorithm designers. To address this problem, we have developed a* semiformal *correctness proof method for the* asynchronous *message passing environment, using ideas from well known* formal *correctness proof methods. In this paper, we illustrate part of the proof method by proving the safety property of a simple network broadcast algorithm.*

1 Introduction

Our main purpose in this paper is to provide a *semiformal* correctness proof of the safety property of a network broadcast algorithm. Below we discuss our motivation for this work in more detail.

In recent years, a large number of published distributed algorithms have been shown to be incorrect. To quote Tanenbaum in [15, page 502], "Worst of all, a large fraction of all the published algorithms in this area [distributed deadlock detection] are just plain wrong, including those proven to be correct... It often occurs that shortly after an algorithm is invented, proven correct, and then published, somebody finds a counterexample". For example, [5] showed that a distributed algorithm [12] for deadlock detection in distributed databases was incorrect. Further examples of erroneous distributed deadlock detection algorithms are given in [7,14]. [8] showed that an algorithm for distributed termination detection [1] would detect false termination. In [9], we showed that a multiprocess synchronization algorithm [2] had a livelock problem. In [11], we showed that the algorithm in [3] to implement the generalized input-output construct of CSP [6] was incorrect. In [10], we showed that the distributed depth first search algorithm of [13] had some minor correctness problems.

Note that the problem cannot be significantly reduced by simply asking the authors and referees to be more careful. Also, known testing methods for sequential algorithms are not well suited for distributed algorithms. *Formal* proof methods are not commonly used by algorithm designers in the published literature for a variety of reasons, including being too mathematical and requiring the designer to rewrite the code in a more abstract notation. *Informal* correctness proofs are too unreliable.

To address the above problems, we devised a *semiformal* proof method. This proof method is more rigorous than the usual *informal* proofs, but simpler and less mathematical than the usual *formal* proofs and does not require the designers to rewrite their algorithm in some abstract programming notation. This proof method assumes the *asynchronous* message passing environment. Therefore hopefully it will be more attractive to common algorithm designers and readers.

Due to space limitations, we cannot discuss the proof method in this paper. Instead, we illustrate the use of the proof method by focusing on a simple network broadcast problem, presenting a simple algorithm for it, and then providing a brief correctness proof of its safety. For the sake of brevity, we skip discussion of the liveness properties of the algorithm. Similarly we skip several details in the safety proof and in the rest of our discussion. Our objective in this paper is to provide a flavor of the proof, not all its details.

2 Programming Environment

We assume that the message communication in the distributed system is *asynchronous.* In other words, a sender process i does not need to synchronize with the receiver process j before i can send a message to j. We do not require the communication channels to be First-In-First-Out (FIFO). A message sent to a process *arrives* at the input port of the process after

0730-3157/97 $10.00

an arbitrary but finite communication delay (possibly zero). A message that has been sent but has not arrived at the input port of the receiver is said to be *in transit.*

The code is written in the style of *guarded commands* (or *rules*) at any process i. The basic idea of guarded commands has been discussed in [4,6]. We use a somewhat different syntax and semantics because of the asynchronous message passing environment.

3 Definition of the Network Broadcast Problem

3.1 Informal Description

Let $G=(V,E)$ be a connected, undirected graph where V denotes the set of vertices, numbered $1,2,\ldots,N(N\geq 1)$, and E denotes the set of undirected edges in the graph. Consider the topologically isomorphic computer network where each node i is represented by the process i, and each edge is represented by a bidirectional communication line. Let T be a particular spanning tree of this graph. Each process i in the network has the following information:

1. The value of its own id i, say in a variable *myid*.
2. The id of the father node in the spanning tree T, say in a variable *father*. If i is the root node of the tree, then the value of *father* will be the same as *myid*.
3. A set *sons* containing the ids of the son nodes of node i in the spanning tree T.
4. The value of N.
5. The value of $mydata_i$. This is a local data at process i that needs to be communicated to all other processes.

The objective of the network broadcast problem considered here is to devise a distributed algorithm so that each process acquires a copy of the various *mydata* values stored at the processes in the network. The first message in the system should be sent by the root; other processes should not send their first message before receiving a message. Informally, we require the following correctness properties of the algorithm:

1. The system computation should *terminate* within a finite time.
2. When the system computation is terminated, the following properties hold: (a) Each process i knows the values of various $mydata_j$ in the system, say in an array A, i.e., $A_i[j]=mydata_j$. (b) Each process is terminated, i.e., it has executed the statement "terminate this process", and (c) there are no (unreceived) messages at the input port of any process.

3.2 Specification of Desired Safety Properties:

Property SF below states the safety property that a correct algorithm must satisfy.

SF. If the system computation is currently terminated, then at this point the following assertions are true: (in SF1-SF3 below, the variables u, w range over integers $1,2,\ldots,N$. The range of other variables is arbitrary, unless stated otherwise. All free variables u, w are universally quantified.)

SF1. $A_u[w] = mydata_w$.
SF2. $terminated_u$ =TRUE.
SF3. $\exists$ no message at the input port of process u.

4 The Algorithm

4.1 Global Constant Declarations

N = The total number of processes in the system; (* $N\geq 1$. *)

4.2 Declaration of Variables Local To Process i

mydata : integer; (* The data owned by process i, to be passed on to every other process. *)

myid : integer; (* Id of this process *)

father : integer; (* id of the father node in the tree T if i is not the root node; otherwise its value is the same as *myid*. *)

sons : set of integers; (* Contains ids of son nodes. As a special case, the set may be empty. *)

neigh : set of integers; (* Contains ids of neighbors of this node along the tree T, i.e., if $father \neq myid$ then $neigh = sons \cup \{father\}$ else $neigh = sons$. This variable is defined in order to avoid computing the above expression several times during the execution. *)

A : array[1... N] of integer; (* $A[j]$, once defined, would contain the *mydata* value of process j. *)

Aentries : integer; (* The number of elements of the array A where actual data values have been stored. *)

in_hand : array[1... N] of boolean; (* $in\_hand[j]$ = TRUE if the value $mydata_j$ has already been stored in $A[j]$, FALSE otherwise. This is an auxiliary variable — needed for the proof only. *)

4.3 Initialization Code Of Process i

myid := i;
initialize *father* and *sons* appropriately;
initialize *mydata* to the appropriate value;
$neigh := sons \cup (\{father\} - \{myid\})$;
(* if $father \neq myid$ then $neigh = sons \cup \{father\}$
 else $neigh = sons$. *)
Aentries := 0;
for j := 1 to N do
    $in\_hand[j]$:= FALSE;
if $father = myid$ then send message
  $M(myid, myid, mydata)$ to process *myid*;

4.4 Guarded Commands (Rules) of Process i

(* Receiving data. *)
S1. received message $M(owner\_id, sender\_id, data) \rightarrow$
    $A[owner\_id] := data$;
    $Aentries := Aentries+1$;
    $in\_hand[owner\_id]$:= TRUE;

X := *neigh*− {*sender_id*};
send message $M(owner\_id, myid, data)$ to each process in X;
if $Aentries = 1 \wedge father \neq myid$ then send message $M(myid, myid, mydata)$ to process *myid*;
(* The process is going to terminate now. *)
S2. $Aentries = N \rightarrow$ terminate this process;

5 Proof of Safety Properties

Corresponding to any node i, let us define a tree $T(i)$ which is identical to the given tree T as an acyclic connected graph but whose root is the node i. For any node j, let $F(j,i)$ denote the father of node j corresponding to the tree $T(i)$. (As in the definition of the variable *father*, which corresponds to the tree T, we have $F(j,i)=j$ for $j=i$). Similarly, let $S(j,i)$ denote the set of sons of node j corresponding to the tree $T(i)$.

Define the assertion I to be the conjunction of assertions A1-A10 below. Later in Lemma 1, we will show that I is an invariant. For the various assertions stated in this paper, we assume that the variables u, w, x range over integers $1,2,\ldots,N$. Range of other variables is arbitrary, unless stated otherwise. All free variables u, w, x in these assertions are universally quantified.

A1. $Aentries_u$ = the number of processes w such that $in\_hand_u[w]$.
A2. $in\_hand_u[w] \Rightarrow A_u[w] = mydata_w$.
A3. $(x = F(u,w) \wedge in\_hand_u[w]) \Rightarrow in\_hand_x[w]$.
A4. given any message at the input port of, or in transit to, process u, this message is of the form $M(w,x,data)$ where $x = F(u,w)$, and $data = mydata_w$.
A5. $x = F(u,w) \Rightarrow$
([$\exists$ a message of the form $M(w,\ldots,\ldots)$ at the input port of, or in transit to, process u] $\Leftrightarrow$
[$(\neg\, in\_hand_u[w]) \wedge$
$([w \neq u \wedge in\_hand_x[w]] \vee [w = u \wedge$
$(father_u = myid_u \;\vee\; Aentries_u > 0)])$
]
).
A6. $\forall w :: \exists$ at most one message of the form $M(w,\ldots,\ldots)$ at the input port of, or in transit to, process u.
(* Note: For a given u, there may be several such messages, but with different values of w.*)
A7. $terminated_u \Rightarrow Aentries_u = N$.
A8. $neigh_u = S(u,w) \cup (\{F(u,w)\} - \{u\})$.
A9. $\exists u :: father_u = myid_u$.
A10. $myid_u = u$.

Lemma 1: The assertion I (i.e., the conjunction of A1-A10) is an invariant.
Proof: Obviously I is true after the initialization. Also, if a transient message arrives at an input port, clearly this event does not change the value of I from true to false. Now we need to show that, for any rule execution at any process i, the rule execution indeed terminates gracefully and I remains true at the end of the rule execution. We do this informally in our proof method; the details are skipped here.

Lemma 2: [System computation has terminated] $\Rightarrow$ [$\forall u :: \exists$ no message at the input port of, or in transit to, process u].

Proof: By definition of the system computation being terminated, there are no transient messages at this point. Now suppose process u has a message at its input port. By A4, this message must be of the form $M(w,x,data)$ where $1 \leq w, x \leq N$, and $x = F(u,w)$. By A5, $in\_hand_u[w]$ is FALSE. Therefore by A1, we have $Aentries_u < N$. Hence by A7, $terminated_u$ is FALSE. Therefore rule S1 at process u is ready. This contradicts the hypothesis that the system computation has terminated.

Lemma 3: [System computation has terminated] $\Rightarrow$ [$\forall u, w, x : x = F(u,w) ::$ $(in\_hand_x[w] \Rightarrow in\_hand_u[w])$].

Proof: Assume that the system computation has terminated, and consider any u,w,x satisfying $x = F(u,w)$. If $u = w$ then $x = u$ and the result obviously holds.

Now consider the case $u \neq w$. Suppose $in\_hand_x[w] \wedge \neg in\_hand_u[w]$ is TRUE. By A5, we conclude that $\exists$ a message $M(w,\ldots,\ldots)$ at the input port of, or in transit to, process u. This contradicts Lemma 2.

Lemma 4: [System computation has terminated] $\Rightarrow$ [$\exists w :: in\_hand_w[w]$].

Proof: By A9, there exists a process w such that $1 \leq w \leq N \wedge father_w = myid_w$. Suppose $in\_hand_w[w]$ is FALSE. By substituting $u = w$ in A5, we conclude that $\exists$ a message $M(w,\ldots,\ldots)$ at the input port of, or in transit to, process w. This contradicts Lemma 2.

Lemma 5: [System computation has terminated] $\Rightarrow$ [$\forall u, w :: in\_hand_w[w] \Rightarrow in\_hand_u[w]$].

Proof: Suppose the system computation has terminated, and consider any w satisfying $in\_hand_w[w]$. We establish $in\_hand_u[w]$ by proceeding inductively on the level of node u in the tree $T(w)$, starting with the root of the tree. The base case is obvious, since in that case we have $u = w$.

Inductively, consider any node u other than node w on the tree. Let $x = F(u,w)$. By the induction hypothesis, we have $in\_hand_x[w]$. By Lemma 3, we obviously get $in\_hand_u[w]$.

Lemma 6: [System computation has terminated] $\Rightarrow$ [$\forall u :: in\_hand_u[u]$].

Proof: Consider any process w stipulated in Lemma 4. Let u be any process. By Lemma 5, we have $in\_hand_u[w]$. We need to show that $in\_hand_u[u]$ is TRUE.

Suppose $in\_hand_u[u]$ is FALSE. Since $in\_hand_u[w]$ is TRUE, from A1 we get $Aentries_u > 0$. Therefore by A5 we conclude that there is a message of the form $M(u,\ldots,\ldots)$ at the input port of, or in transit to, process u. This contradicts Lemma 2.

Lemma 7: [System computation has terminated] $\Rightarrow$ $[\forall u, w :: in\_hand_u[w]]$.

Proof: Follows immediately from Lemmas 6 and 5.

Theorem 1: Our algorithm satisfies the safety property SF stated in section 3.2.

Proof: Suppose the system computation has terminated.

SF1: The result immediately follows from Lemma 7, A2, and Lemma 1.

SF2: Consider any process u. From Lemma 7, A1, and Lemma 1, we get $Aentries_u = N$. Obviously $terminated_u$ must be TRUE, because otherwise the rule S2 at process u would be ready, contradicting the hypothesis that the system computation has terminated.

SF3: Follows immediately from Lemma 2.

6 Concluding Remarks

In comparison with formal proof methods, our method is simpler to understand and apply, in many respects:

1. Our language directly employs asynchronous message passing, so the algorithm designer does not have to rewrite the algorithm in a very different programming environment.
2. We have tried to keep our programming notation close to the more common way in which algorithms are presented in the literature. Thus we use a guarded command based notation and avoid mathematical looking notation.
3. Our programming language is fairly simple, resulting in a simpler proof method. For example, we don't use nested guarded commands or message reception on the RHS of a guarded command.
4. Delegating part of the proof to informal reasoning further simplifies the proof, without hurting the reliability of the proof much. Note that this informal reasoning does not rely on the reader's understanding of the overall algorithm; it only relies upon his understanding of the underlying programming language used. Thus it does not significantly hurt the rigor or reliability of the proof.

In comparison with the usual informal proofs, our proof is far more reliable. In particular, it forces the authors to explicitly specify essential properties of all possible reachable global states. This is where a large number of errors occur in informal proofs, since typically only a part of such properties is explicitly stated and the rest is left to intuition — thus it does not form a complete invariant. Moreover, the stated properties are not collected together in one place in papers that are based on informal proofs.

Acknowledgments

The research work of S. S. Iyengar was supported by the Office of Naval Research grant N00014-94-1-0343.

REFERENCES

[1] R. K. Arora, S.P. Rana, and M. N. Gupta, "Distributed Termination Detection Algorithm For Distributed Computations", *Information Processing Letters*, vol. 22, no. 6, pp. 311-314, 1986.

[2] R. Bagrodia, "A Distributed Algorithm for N-Party Interactions", *Technical Report* STP-053-85, Micro-electronics and Computer Corporation (MCC), Austin, Texas, August 1985.

[3] G. N. Buckley and A. Silberschatz, "An effective implementation for the generalized input-output construct of CSP," *ACM Trans. Program. Lang. Syst.*, vol. 5, no. 2, pp. 223-235, 1983.

[4] Dijkstra, E. W., "Guarded commands, nondeterminacy and formal derivation of programs," *CACM* 18, 8, pp. 453-457, August 1975.

[5] V. Gligor and S. Shattuck, "On Deadlock Detection in Distributed Systems, *IEEE-TSE*, Vol. SE-6, No. 5, pp. 435-440, September 1980.

[6] C. A. R. Hoare, "Communicating Sequential Processes", *Communications of the ACM*, Vol. 21, No. 8, pp. 666-777, August 1978.

[7] E. Knapp, "Deadlock Detection in Distributed Databases", *Computing Surveys*, vol. 19, pp. 303-328, Dec. 1987.

[8] D. Kumar, "On The Correctness Of A Termination Detection Algorithm", *Technical Report* TR-87-08, Department of Computer Sciences, University of Texas, Austin, March 1987.

[9] D. Kumar, "An Algorithm for N-Party Synchronization Using Tokens", 10^{th} *International Conference on Distributed Computing Systems*, pp. 320-327, Paris, France, May 28 - June 1, 1990.

[10] D. Kumar, S. S. Iyengar, and M. B. Sharma, "Corrections to a distributed depth-first-search algorithm", *Information Processing Letters*, vol. 35, pp. 55-56, 15 June, 1990.

[11] D. Kumar and A. Silberschatz, "A Counter Example To An Algorithm For The Generalized Input-Output Construct Of CSP", *Information Processing Letters*, April 1997.

[12] D. Menasce and R. Muntz, "Locking and deadlock detection in distributed data bases", *IEEE Transaction on Software Engineering*, vol. 5, no. 3, pp. 195-202, May 1979.

[13] M. B. Sharma, S. S. Iyengar, and N. K. Mandyam, "An Efficient Distributed Depth-First-Search Algorithm", *Information Processing Letters*, vol. 32, pp. 183-186, 1989.

[14] M. Singhal, "Deadlock Detection in Distributed Systems", *IEEE Computer*, vol. 22, pp. 37-48, Nov. 1989.

[15] A. S. Tanenbaum, *Modern Operating Systems*, Prentice Hall, Englewood Cliffs, N.J. 07632, 1992.