Distrib Comput (1995) 9: 147–156

DISTRIBUTED COMPUTING

Optimal asynchronous agreement and leader election algorithm for complete networks with Byzantine faulty links

Hasan M. Sayeed[1], **Marwan Abu-Amara**[2], **Hosame Abu-Amara**[3]

[1]Department of Electrical Engineering, Texas A&M University, College Station, TX 77843, USA
[2]Bell-Northern Research, P.O. Box 833871, Mail Stop D208, Richardson, TX 75083, USA
[3]Department of Electrical and Computer Engineering, University of Nevada, 4505 Maryland Parkway, Las Vegas, NV 89154, USA

Received: May 1994 / Accepted: July 1995

Summary. We consider agreement and leader election on asynchronous complete networks when the processors are reliable, but some of the channels are subject to failure. Fischer, Lynch, and Paterson have already shown that no deterministic algorithm can solve the agreement problem on asynchronous networks if any processor fails during the execution of the algorithm. Therefore, we consider only channel failures. The type of channel failure we consider in this paper is *Byzantine* failure, that is, channels fail by altering messages, sending false information, forging messages, losing messages at will, and so on. There are no restrictions on the behavior of a faulty channel. Therefore, a faulty channel may act as an adversary who forges messages on purpose to prevent the successful completion of the algorithm. Because we assume an asynchronous network, the channel delays are arbitrary. Thus, the faulty channels may not be detectable unless, for example, the faulty channels cause garbage to be sent. We present the first known agreement and leader election algorithm for asynchronous complete networks in which the processors are reliable but some channels may be Byzantine faulty. The algorithm can tolerate up to $\left\lfloor \frac{n-2}{2} \right\rfloor$ faulty channels, where n is the number of processors in the network. We show that the bound on the number of faulty channels is optimal. When the processors terminate their corresponding algorithms, all the processors in the network will have the same correct vector, where the vector contains the private values of all the processors.

Key words: Distributed algorithms – Byzantine agreement – Faulty channels – Asynchronous networks – Fault-tolerant computing

Correspondence to: H. Abu-Amara
e-mail: amara@eesunl.tamu.edu

1 Introduction and literature survey

1.1 Introduction

Reaching agreement among remote processors in a distributed system is one of the most fundamental problems in distributed computing and is at the core of many algorithms for distributed data processing, distributed file management, and fault-tolerant distributed applications [13]. Consider a distributed system of n processors, where each processor u is running a process that computes some private value $ID(u)$. By exchanging messages, each processor u can obtain a vector of the private values computed by all n processors. Processor u can then apply some averaging function on the vector to obtain a number VAL that all processors agree on. All processors then can use VAL to update a common variable. This is useful, for instance, in distributed database systems where several processors may wish to make different alterations in the same file. In order to avoid inconsistencies in the file, the processors exchange messages to agree on which alteration to incorporate in the file. Other applications of agreement include concurrency control, regeneration of a lost (unique) token, recovery by electing a new coordinator after a crash of a coordinator in a distributed database system, and replacing a primary site in a replicated distributed file system [1, 4, 21, 22].

A network is *complete* if every processor has a direct communication channel to every other processor. We consider the agreement problem on asynchronous complete networks when the processors are reliable, but some of the channels are subject to failure. We wish to construct an algorithm such that all n processors will have the same correct vector when the processors terminate their corresponding algorithm. (This is also called achieving *interactive consistency* among the processors [24]). Thus, the processors can use the correct vector to compute a value. Fischer, Lynch, and Paterson [13] have already shown that no algorithm can solve the agreement problem on asynchronous networks if any processor fails during the execution of the algorithm. Fischer, Lynch, and Paterson's impossibility result holds even for fail-stop failure, the most benign of all faults. Therefore, we consider only channel failures.

The type of channel failure we consider in this paper is *Byzantine* failure, that is, channels fail by altering messages, sending false information, forging messages, losing messages at will, and so on [18, 23, 24]. There are no restrictions on the behavior of a faulty channel. Therefore, Byzantine faulty channels act as malicious agents who inject faulty messages in the network to prevent the successful completion of algorithms. Because we assume an asynchronous network, all channel delays are arbitrary. Thus, the faulty channels may not be detectable unless, for example, the faulty channels cause garbage to be sent. Note that a channel can fail at any time during the execution of the algorithm.

In this paper, we present the first known agreement and leader election algorithm for asynchronous complete networks in which the processors are reliable but some channels may be Byzantine faulty. The algorithm can tolerate up to $\left\lfloor \frac{n-2}{2} \right\rfloor$ faulty channels, where n is the number of processors in the network. Throughout the paper, we assume that there are t faulty channels, and that the total number n of processors in the network is at least $2t + 2$. As we show in Theorem 1, if a network may have up to t Byzantine faulty links, then all the faulty links may be incident on one node, and, therefore, agreement will be possible only when the connectivity of the network is at least $2t + 1$. Hence, the bound on the number of processors and faulty channels in our algorithm is optimal. The total number of bits that the nodes send is $O(n^3 \text{ S})$ bits, while the amount of storage that the algorithm uses in each node is $O(n^2 \text{ S})$ bits, where S is the maximum number of bits needed to specify the node private values.

1.2 Literature survey

In the literature, three different, but related, problems were addressed: consensus [6, 9, 11, 12, 13, 19], agreement or interactive consistency [7, 10, 24], and leader election [2, 3, 5, 8, 15, 16, 17, 20, 25]. In the *consensus* problem, all of the non-faulty processors in the network want to agree on a single bit, a 0 or a 1. In the *agreement* problem, each processor in the network has a private value that should be communicated to the other nodes in the network. Thus, at the end of an execution of an algorithm that solves the agreement problem, each non-faulty processor computes a vector with an element for each processor in the network, where the vector computed by all non-faulty processors must be the same, and each vector element that corresponds to a non-faulty processor is the private value of that processor. In the *leader election* problem, all of the processors in the network want to choose a unique processor to be the leader. To solve the election problem, deterministic leader election algorithms assume that there exists at least one unique private value. Note that if a processor, say x, has a Byzantine failure, then the leader election problem cannot be solved simply because x can become the leader, or x can force the other processors to elect a non existent processor. We think of the agreement problem as a generalization of the consensus problem because, after computing the vector in the agreement problem, the non-faulty processors can agree on a particular value. Also, in the agreement problem, if the private value of some processor u is unique, then the leader election problem can be solved, provided that there are no Byzantine faulty processors in the network, simply by having each processor choose the processor with the largest unique private value as the leader.

2 Formal model and definitions

2.1 Formal model

This section discusses the model of the distributed network. Our model follows Goldreich and Shrira's model [16]. Consider a network of n processors. We model the network as a graph of n nodes, in which each node represents a processor, and each link represents a bidirectional communication channel. We will use the term *node* to indicate a processor, and the term *link* to indicate a communication channel. We assume that the network is complete, that is, every node is connected to every other node by a bidirectional communication channel. We also assume that the network is asynchronous, that is, there is no global clock in the system, and each node may have its own clock. The nodes do not have shared memory; they communicate by sending messages to each other on the communication channels between them.

All the nodes are identical except that each node u has a private unique value (identifier) $ID(u)$ chosen from a totally ordered set. Initially, no node knows the identifier of any other node. Each node u knows the number of nodes in the distributed system.

A *distributed algorithm* on a network is a set of n deterministic local programs, each assigned to a node. Each local program consists of *computation statements* and *communication statements*. The *computation statements* control the internal operations of a node. The *communication statements* are of the form "send message M on link l" or "receive message M^* on link l". Each node u has a Send-Buffer(u, l) and a Receive-Buffer(u, l) associated with each link l incident on u, where the buffers are not necessarily first-in-first-out. Let l be the link that connects nodes u and v. When u wishes to send a message M on link l, node u places M in Send-Buffer(u, l). We call this event a *send event*. To capture the asynchronous nature of our network, messages may remain in the send-buffers for arbitrary lengths of time. A *transmission event* in l occurs when l places M in Receive-Buffer(v, l). We assume that u cannot inspect Send-Buffer(u, l) to check whether M was removed from the buffer. Hence, M is *in transit* from u to v if M is in Send-Buffer(u, l). If u wishes to process a message M^* on l, the u removes M^* from Receive-Buffer(u, l). We call this event a *receive event*. If M^* is not in Receive-Buffer(u, l), then u either waits for M^*, or u receives some other message depending on u's local program. When we say that node u *receives* a message, we mean that u *removes* the message from a Receive-Buffer and *processes* the message. A *failure event* in a link l is the event of l spontaneously discarding a message, generating a message that was not created by a processor, or changing the contents of a message that passes through the link. We model each processor u as an automaton. Upon receiving a message M, processor u either *accepts* or *discards* M, depending on u's local program. If u accepts M, then u atomically updates the variables and counters associated with the processor and changes u's state. If u discards M, then u does not change u's state and does not update any variable or counter. Node u always discards damaged messages that arrive at u.

In keeping with current assumptions in the literature [1, 5, 8], a link l is *faulty* during a particular execution E of the algorithm if l experiences at least one failure event in E. If a link l is not faulty, then it is *reliable*. Once a link has been labeled 'faulty', the link continues to be called faulty even if the link is repaired. In other words, a reliable link is a link that never experiences a failure event [1, 5, 8]. A bidirectional link is considered faulty even if the link is faulty in only one direction. In other words, if a link l connects nodes u and v, and l experiences failure events when messages traverse from u to v, but not from v to u, we still consider l to be faulty in both directions. We assume that, in any execution of the algorithm, there can be no more than t faulty links, and $n \geqq 2t + 2$.

Consider a particular execution E of a distributed algorithm. For convenience, we assume the existence of a global clock that gives the time at which each event in E occurs. Although this clock is available to an observer of the network, the nodes do not know of its existence. We will assume that each event in E occurs at some discrete unit of time starting from zero. Let *Events*(u) be the multiset of u's send and receive events in E. The local program in u induces a total ordering on *Events*(u). Two events, each in a distinct node, may occur at the same time. However, two events cannot occur at the same time in the same node.

We assume that, when a node receives a message, then the node knows on which link the message was received. The delay on a reliable link is arbitrary but finite. Thus, messages sent on reliable links must be eventually delivered. Because of the asynchronous nature of the network, a node cannot distinguish between a slow link and a faulty link. Therefore, the faulty links may not be detectable unless, for example, the faulty links cause garbage to be sent. Links may fail at any time before or during the execution of the algorithm. All the nodes in the network are reliable. It is not necessary that all nodes start the execution of the algorithm simultaneously; some node may be initially dormant. We assume that, if a dormant node receives a message from some other node, then the dormant node wakes up and starts executing the algorithm.

2.2 Definitions

– A processor k receives a *correct message* M from another processor r if M is an exact copy of the original message sent by r. Otherwise, the message M is *faulty*.

– Consider a network of n processors, where each processor has some private value. Consider some processor k in the network. Let Vector(k) be a vector formed by k from the n values $V_1^k, V_2^k, \ldots, V_n^k$, that k receives from all the processors in the network (including k). Thus, we write Vector(k) $= (V_1^k, V_2^k, \ldots, V_n^k)$. We say that Vector($k$) is *correct* if, for all $l \leqq i \leqq n$, the value V_i^k is equal to the private value held by processor i. Otherwise, Vector(k) is said to be *faulty*.

– Consider any two processors, say r and s, in a network on n processors. Let the two vectors formed by r and s be Vector(r) $= (V_1^r, V_2^r, \ldots, V_n^r)$ and Vector(s) $= (V_1^s, V_2^s, \ldots, V_n^s)$, respectively. Then, the two vectors Vector(r) and Vector(s) are said to be *identical* if $V_i^r = V_i^s$, for all $1 \leqq i \leqq n$. Furthermore, if Vector(r) and Vector(s) are identical and correct, then we say that r and s have computed the *same correct vector*.

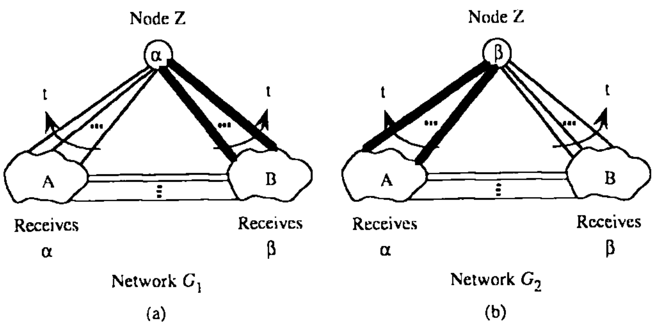

Fig. 1a, b. Two complete networks, where a bold line denotes a faulty link

3 Lower bound for connectivity

The lower bound proof presented below is similar to the connectivity lower bound proof for synchronous networks with Byzantine faulty *processors* shown in Lemma 5 of [10]. We assume that the algorithms are "full information" algorithms [14].

Theorem 1. *Byzantine agreement is not possible in complete networks that have t faulty links and a connectivity of at most* 2t.

Proof. Consider the two networks, G_1 and G_2, shown in Fig. 1. Let the IDs of all the nodes of networks G_1 and G_2 be identical except for node Z, as shown in Fig. 1. Assume, contrary to the theorem, there exists an algorithm Π that ensures Byzantine agreement in complete networks that have t faulty links and a connectivity of $2t$. Thus, Π solves Byzantine agreement on both G_1 and G_2. In Fig. 1a, subnetwork A receives α as the ID of node Z, whereas subnetwork B receives β as the ID of node Z because B is connected to Z through t faulty links. In Fig. 1a, the faulty links follow the doctrine [10]: substitute the value α for β in every message that passes from B to Z, and substitute the value β for α in the messages passing back from Z to B. Similarly, in Fig. 1b, subnetwork A receives α as the ID of node Z because A is connected to Z through t faulty links, whereas subnetwork B receives β as the ID of node Z. In Fig. 1b, the faulty links follow the doctrine: substitute the value β for α in every message that passes from A to Z, and substitute the value α for β in the messages passing back from Z to A. Hence, subnetworks A and B of both G_1 and G_2 have the same inputs to Π. Thus, the final vector computed by the nodes in subnetworks A and B of both G_1

and G_2 after running Π will be identical. However, according to the definition of Byzantine agreement, and since G_1 has α as the ID for node Z and G_2 has β as the ID for node Z, the nodes in subnetworks A and B of G_1 should compute a final vector that differs from the final vector computed by the nodes in subnetworks A and B of G_2. A contradiction. □

4 Algorithm

4.1 Intuition and description of the algorithm

Appendix A has a detailed description of our algorithm. We say that an ID γ is *faulty* if γ was fabricated by a faulty link and is not the ID of a node. A node *broadcasts* a message if the node sends the message to all its neighbors. In a nutshell, the algorithm consists of two parts: In the first part, each node u broadcasts ID(u) in an "Announce-ID" message, and upon receiving such a message M, u forwards M to u's neighbors in a "Verify-ID" message. In the second part, each node that computes a vector V consisting of the correct identifiers of all nodes broadcasts a "Correct-Final, V" message, and each node that receives two "Correct-Final, V" messages (from two distinct nodes) broadcasts a "Verify-Final, V" message. A node decides on a vector V if the node computes directly that V is the correct vector or if the node receives $t + 1$ "Correct-Final, V" and "Verify-Final, V" messages.

Our algorithm has three properties:

liveness: At least two nodes broadcast "Correct-Final, V" messages, where V is the correct vector of identifiers,
resiliency: a node broadcasts a "Correct-Final, V" message only when V is the correct vector of IDs, and
progression: if a node receives two "Verify-Final, V" messages on some two distinct links l and l^*, and in addition the node receives "Verify-Final, V" or "Correct-Final, V" messages on at least $t - 1$ distinct links different from l and l^*, then V is the correct vector of IDs.

We first explain how the algorithm ensures that the progression property is true. Assume that the resiliency property holds for the algorithm. Since there are at most t faulty links in the network, and there are at least $2t + 2$ nodes in the network, there are at least two nodes P and Q that are not adjacent to any faulty links. Our algorithm ensures that nodes P and Q compute the correct final vector V. Nodes P and Q, then, broadcast the message "Correct-Final, V" to all other nodes, and P and Q stop executing the algorithm. Since all nodes are connected to P and Q via non-faulty links, all nodes receive the Correct-Final messages from P and Q.

Since nodes do not initially know which links are faulty, and faulty links may fabricate Correct-Final messages that contain faulty vectors, each node Y, upon receiving two "Correct-Final, V" messages on some two distinct links r and r^*, broadcasts the message, "Verify-Final, V" to all other nodes. A node that receives a message "Verify-Final, V" does not relay it to other nodes. Suppose that Y receives two "Correct-Final, V*" messages from two distinct links l and l^*, for some vector V* of IDs. Then, we claim that at least $t - 1$ links in addition to l and l^* deliver

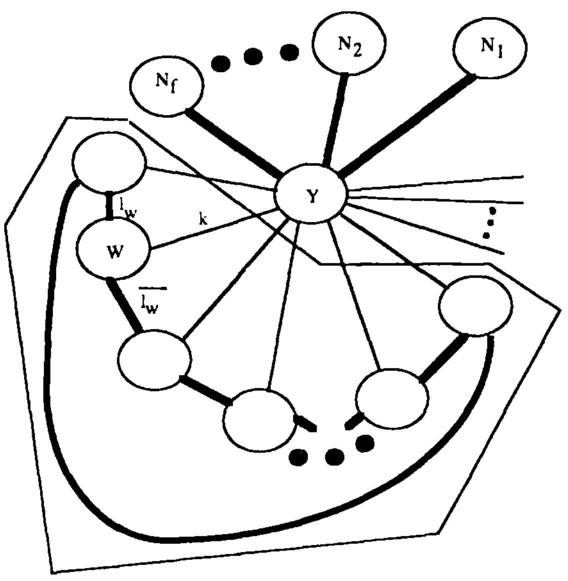

Fig. 2. Nodes Y and W and the adjacent faulty links

"Correct-Final, V*" or "Verify-Final, V*" messages to Y if, and only if, V* is the correct final vector. Hence, when Y receives "Correct-Final, V" or "Verify-Final, V" from at least $t - 1$ links in addition to r and r^*, node Y concludes that "Correct-Final, V" is an authentic message, takes V as the correct final vector, broadcasts "Correct-Final, V", and stops executing the algorithm.

To see why at least $t - 1$ links in addition to l and l^* deliver "Correct-Final, V*" or "Verify-Final, V*" messages to Y if V* is the correct final vector, recall that P and Q broadcast the message "Correct-Final, V" with V = V*. Hence, all nodes, other than P, Q, and Y, will broadcast "Correct-Final, V*" or "Verify-Final, V*" messages. Some of these messages may be lost in faulty links that may be adjacent to Y. Nevertheless, Y will receive such messages on at least (number of nodes without P, Q, and Y) $- t = (n - 3) - t \geqq (2t + 2 - 3) - t = t - 1$ links.

On the other hand, we claim at most $t - 2$ links in addition to l and l^* deliver "Correct-Final, V*" or "Verify-Final, V*" messages to Y if V* is not the correct final vector. If V* is not the correct final vector, and Y receives "Correct-Final, V*" on some link k^*, then k^* must be faulty, by the resiliency property of our algorithm. Hence, l and l^* are both faulty. On the other hand, if Y receives "Verify-Final, V*" on some link k, then k can be non-faulty. This can happen, for example, in the situation illustrated in Fig. 2, where the bold lines indicate faulty links. The non-faulty link k connects Y to node W, and there are two faulty links l_w and $\overline{l_w}$ incident on W. If l_w and $\overline{l_w}$ fabricate the faulty message "Correct-Final, V*", then W may send the message "Verify-Final, V*" on link k. Hence, let Final-naive(Y) be the set of nodes that are connected to Y via non-faulty links and send "Verify-Final, V*" to Y. Then W is a member of Final-naive(Y), and each member of Final-naive(Y) is adjacent to at least two faulty links. Suppose that there are f faulty links incident on Y (including l and l^*). Since there are at most t faulty links in the network, then Final-naive(Y) contains at most $t - f$ nodes, as shown in Fig. 2, where the nodes in

Final-naive(Y) are contained in the polygon. Hence, at most $f - 2$ faulty links incident on Y (other than l and l^*) deliver "Correct-Final, V*" or "Verify-Final, V*" messages to Y, and at most $t - f$ non-faulty links that connect Final-naive(Y) nodes to Y deliver "Verify-Final, V*" messages to Y. Hence, at most $f - 2 + t - f = t - 2$ links in addition to l and l^* deliver "Correct-Final, V*" or "Verify-Final, V*" messages to Y.

The difficult part in the design of our algorithm is ensuring the resiliency property. In the algorithm, some nodes broadcast Correct-Final messages because the nodes correctly collect all the n node IDs, while other nodes, e.g. Y above, broadcast a Correct-Final message because they received Correct-Final and Verify-Final messages. We call the former type of nodes *prime nodes*, while we call the latter type *secondary nodes*. From our previous description of how Y responds to Correct-Final messages, we see that if prime nodes maintain the resiliency property, then secondary nodes will also maintain the resiliency property. To maintain the resiliency property for prime nodes, our algorithm uses two types of messages: Announce-ID and Verify-ID, as follows. When a node W starts executing the algorithm, W broadcasts the message "Announce-ID, ID(W)" to all nodes. Because faulty links may fabricate Announce-ID messages, each node X that receives "Announce-ID, ID(W)" on some link l_x does one of the following, depending on the sequence of messages that X accepted prior to "Announce ID, ID(W)". (Recall that X *accepts* a message if X does not discard the message.)

If X has already accepted an Announce-ID message on l_x, then X concludes that l_x is faulty, discards the message "Announce-ID, ID(W)" that arrived on l_x, and discards all subsequent messages that arrive on l_x. When we say that X discards a message, we mean that X behaves as if X never received the message. Hence, discarded messages never change the internal variables and counters contained in X. In other words, X accepts at most one Announce-ID message from each link incident on X.

If X has already accepted a message "Announce-ID, ID(W)" on some link $\overline{l_x}$ different from l_x, then X does not know which of $\overline{l_x}$ and l_x is faulty. Further, X does not know whether W is connected to X by $\overline{l_x}$ or by l_x. We observe, however, that the problem definition does not require X to know which ID belongs to what node. The problem definition requires X to know only the vector of all node IDs. Hence, X in our algorithm arbitrarily assumes that W is connected to X via link $\overline{l_x}$ and not via link l_x, simply because X accepted the message "Announce-ID, ID(W)" on $\overline{l_x}$ before X received the similar message on l_x. This assumption may be incorrect, but it does not affect the correctness of the algorithm, as we show in the proof of correctness. Hence, X discards the message "Announce-ID, ID(W)" that X received on l_x, but X continues to receive subsequent messages from l_x.

Finally, if the message "Announce-ID, ID(W)" that X received on l_x is the first Announce-ID message that X received on l_x, and X never previously accepted a message "Announce-ID, ID(W)" on any link incident on X, then X tries to verify whether there indeed exists a node

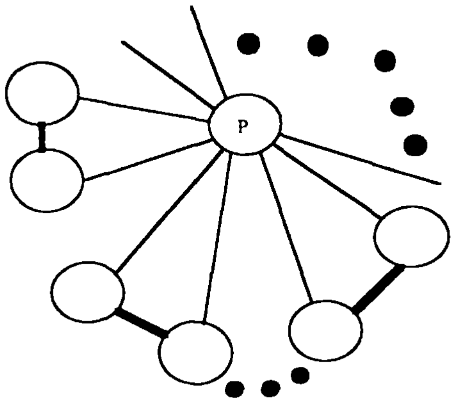

Fig. 3. Node P

with an identifier ID(W). It is this verification that ensures resiliency for prime nodes, and it forms the central part of our algorithm. Once X has verified n IDs, then X becomes a prime node, broadcasts a Correct-Final message that contains all the n verified IDs, and stops executing the algorithm. Node X verifies that there indeed exists a node with identifier ID(W), as follows.

Node X broadcasts the message "Verify-ID, ID(W)" to all other nodes. A node that receives a message "Verify-ID, ID(W)" does not relay it to other nodes. (By default and for completeness, after W broadcasts the "Announce-ID, ID(W)" message, W immediately broadcasts a "Verify-ID, ID(W)" message to all nodes). Node X considers as *certified* the IDs in the Announce-ID messages accepted by X. Hence, X considers ID(W) certified. Note that a certified ID can be a faulty ID because a faulty link may fabricate an Announce-ID message. Nevertheless, X can certify at most n IDs because X accepts at most one Announce-ID message from each link incident on X. (By default, X considers ID(X) as certified.) To determine whether ID(W) is faulty, X maintains a counter CNT_Sum(X) that counts the number of all Announce-ID and Verify-ID messages that X accepted for all certified IDs at X. Node X discards a message M that arrives from some link α if X previously received an identical message to M on α. Hence, X increments CNT_Sum(X) for each "Verify-ID, ID(W)" message that X accepts. Node X waits until X certifies n IDs and CNT_Sum(X) becomes at least $n^2 - 2t$. This waiting does not lead to a deadlock in the network because there exists at least two nodes P and Q that are not adjacent to any faulty links. Hence, P and Q will certify n IDs, (which will be the node IDs). As in Fig. 3, and Lemma 4, each of P and Q receives and accepts at least $n(n - 1) - 2t$ Verify-ID messages for the n certified IDs, where t faulty links may or may not deliver Announce-ID messages. Hence, CNT_Sum(P) and CNT_Sum(Q) will be at least $n(n - 1) - 2t + n = n^2 - 2t$, and the waiting does not cause a deadlock. Thus, suppose that X certifies n IDs and CNT_Sum(X) becomes at least $n^2 - 2t$. It is difficult for X to determine which of the n certified IDs is faulty. Node X, however, can determine whether *no* certified ID in X is faulty, as follows.

First, X uses the messages that X has accepted so far to attempt to compute the number Suspect_Count(X) of faulty links in the network, as follows. For each certified ID γ in X, let link(γ) be the link from which X accepted "Announce-ID, γ". For each two certified IDs γ_1 and γ_2, node X assumes that there is a possibly faulty link between γ_1 and γ_2 if X does not receive "Verify-ID, γ_2" from link(γ_1) or if X does not receive "Verify-ID, γ_1" from link(γ_2). Second X compares Suspect_Count(X) with the value t. By using a counting argument, Lemma 2 proves that, if Suspect_Count(X) is smaller than or equal to t, then none of the n certified IDs in X is faulty. In this case, X forms the final vector V from the n certified IDs in X, broadcasts the message "Correct-Final, V" to all other nodes, and stops executing the algorithm. On the other hand, X may compute a Suspect_Count(X) that is greater than t.

Lemma 2 shows that one reason for this is that one of the certified IDs in X is faulty. Asynchrony and unpredictable message delays is another reason that may cause X to compute a Suspect_Count(X) that is greater than t. Consider, for example, the situation where X assumes that there is a possibly faulty link between γ_1 and γ_2 because X did not receive "Verify-ID, γ_2" from link(γ_1). The link between γ_1 and γ_2 may in fact be reliable, but link(γ_1) may be very slow, and link(γ_1) did not yet deliver the message "Verify-ID, γ_2" to X when X computed Suspect_Count(X). This can cause Suspect_Count(X) to be greater than t. Since X does not know why Suspect_Count(X) is greater than t, node X simply continues to receive and respond to messages. Whenever CNT_Sum(X) increases, X recomputes Suspect_Count(X) and compares it against t. This procedure continues until Suspect_Count(X) becomes at most t, or until X becomes a secondary node.

The algorithm ensures that the liveness property is true as follows. Lemma 4 shows that, if no two nodes other than P and Q become prime nodes, then, since nodes P and Q are not adjacent to any faulty links, there exists a time after which Suspect_Count(P) and Suspect_Count(Q) will be at most t, and P and Q will become prime nodes.

There is one minor technical detail in the description of the algorithm that may cause confusion. Recall that, after a node X in our algorithm announces the ID of X by broadcasting the "Announce-ID, ID(X)" message, node X also broadcasts the "Verify-ID, ID(X)" message. This detail was included to ensure that the various counters are set properly. On the other hand, after a node broadcasts a Correct-Final message, the node does not broadcast any more messages of any kind.

To reduce the communication complexity, our algorithm allows each node to broadcast at most one Verify-Final message. Lemma 5 will show that this restriction on the nodes will not affect the correctness of the algorithm.

4.2 Counters in the algorithm

In our description of the algorithm, some of the messages contain only *one* ID, while others contain a *vector* of n IDs. The messages that contain only *one* ID are of type Announce-ID or type Verify-ID. For each link l incident on Y, Y creates a list IDMessage-List(Y, l) of all Announce-ID and Verify-ID messages that Y accepted

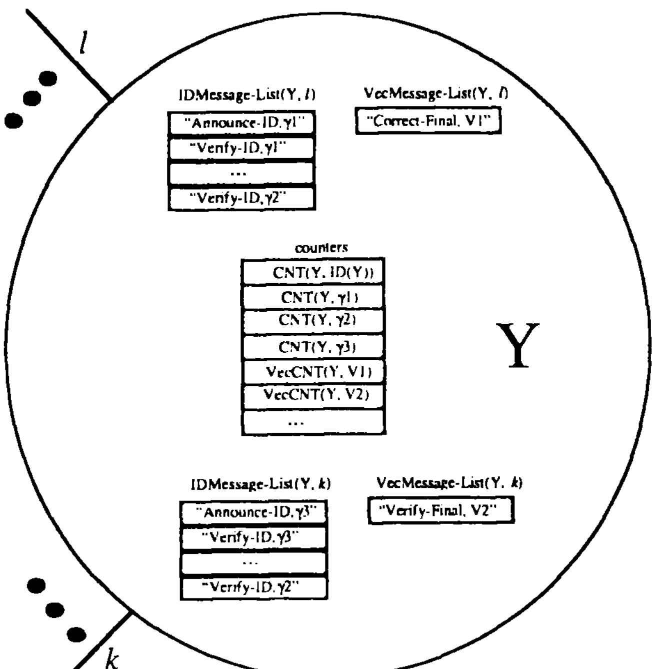

Fig. 4. Counters and lists in node Y

from l. Since there are exactly $n-1$ links incident on Y, node Y creates at most $n-1$ IDMessage-Lists. For each ID β in an Announce-ID or Verify-ID message accepted by Y, Y associates a counter CNT(Y, β) with β, where the initial value of the counter is one. Node Y increments CNT(Y, β) by one for each subsequent Announce-ID or Verify-ID message containing β and accepted by Y. By default, Y creates a counter CNT(Y, ID(Y)) after Y broadcasts the message "Announce-ID, ID(Y)", where the initial value of the counter is one.

On the other hand, the messages that contain a *vector* of n IDs are of type Correct-Final or type Verify-Final. For each link l incident on Y, Y creates a list VecMessage-List(Y, l) of all Correct-Final and Verify-Final messages that Y accepted from l. Hence, node Y creates at most $n-1$ VecMessage-Lists. For each vector V of IDs in a Correct-Final or Verify-Final message accepted by Y, Y associates a counter VecCNT(Y, V) with V, where the initial value of the counter is one. Node Y increments VecCNT(Y, V) by one for each subsequent Correct-Final or Verify-Final message containing V and accepted by Y. Figure 4 shows the counters and lists in node Y. As we argue in Theorem 4, Y creates at most n^2 CNT and $(n-1)$ VecCNT counters.

In the algorithm, Y discards a message M that arrives from some link α if Y previously received an identical message to M on α. Also, if IDMessage-List(Y, l) contains exactly $n+1$ messages for some link l incident on Y, then Y discards all subsequent Announce-ID and Verify-ID messages that arrive on l. Similarly, if VecMessage-List(Y, k) contains exactly one message, say M, for some link k incident on Y, then Y discards all subsequent Correct-Final and Verify-Final messages that arrive on k.

5 Proof of correctness

Lemma 1. *The value of each* CNT *and each* VecCNT *counter in each node never exceeds the value n.*

Proof. By the algorithm, for each CNT(Y, γ) in each node Y, Y accepts only one "Announce-ID, γ" message. Also, for each link incident on Y, Y accepts only one "Verify-ID, γ" message. Hence, the lemma will be true.

By the algorithm, for each VecCNT(Y, V) in each node Y, each link incident on Y contributes at most one message towards incrementing VecCNT(Y, V). Hence, the lemma is true. □

Suppose that a node Y certifies the n IDs contained in the set CERTIFY(Y). Let CNT_Sum(Y) be at least $n^2 - 2t$ at some time T_0. The following lemma shows that the algorithm ensures that the resiliency property is true for prime nodes. Lemma 3 will then show that the resiliency property is true for all nodes.

Lemma 2. *If* CERTIFY(Y) *contains a faulty* ID, *then* Suspect_Count(Y) *at time* T *is greater than t, for every time* $T \geqq T_0$.

Proof. Suppose that CERTIFY(Y) contains the faulty ID δ. Then there is some correct ID γ that is not in CERTIFY(Y). Let Z be the node whose ID is γ. The proof of the lemma proceeds as follows:

Let ID-naive(Y, T, δ) be the set of nodes {X | the set of IDs received by Y at or before time T on the link (X, Y) in Announce-ID or Verify-ID messages is exactly CERTIFY(Y)}. Then, for each X in ID-naive(Y, T, δ), at least one of the links (X, Y) or (X, Z) is faulty. (Otherwise, the message "Verify-ID, γ" must be among the n Verify-ID messages received by Y on link (X, Y)).

Let ID-robust(Y, T, δ) be the set of nodes {X | X $\neq$ Y and the ID δ was not received by Y at or before time T on the link (X, Y) in Announce-ID or Verify-ID messages}. Also, let Additional-naive(Y, T, δ) be the complement of [ID-naive(Y, T, δ)$\cup$ID-robust(Y, T, δ)$\cup${Y}].

Suppose that N = size of ID-naive(Y, T, δ) = |ID-naive(Y, T, δ)|, A = |Additional-naive(Y, T, δ)|, and R = |ID-robust(Y, T, δ)|. Then, $N + A + R = n - 1$.

For each Additional-naive(Y, T, δ) node W, there exists at least one certified ID β in Y such that IDMessage-List(Y, k) does not contain "Verify-ID, β", where k connects W to Y. In the worst case, β can be the ID of another Additional-naive(Y, T, δ). Hence, Y computes at least $\frac{A}{2}$ possibly faulty links incident on the Additional-navie(Y, T, δ) nodes. Also, since Y does not receive the ID δ from the nodes in ID-robust(Y, T, δ), Y computes at least R possibly faulty links incident on the ID-robust(Y, T, δ) nodes. Thus, Suspect_Count(Y) $\geqq R + \frac{A}{2} = R + \frac{n-1-R-N}{2} = \frac{n-1+R+N}{2}$.

Note that ID-robust(Y, T, δ) includes all the nodes that are not adjacent to any faulty links, and the number of these nodes is at least $n - N - 1 - 2(t - N)$. Thus, $R \geqq n + N - 1 - 2t$. Hence, Suspect_Count(Y) $\geqq \frac{n-1+R-N}{2} \geqq n - 1 - t > t$ because $n \geqq 2t + 2$. □

Lemma 3. *Suppose that* V *is an* ID *vector that contains a faulty* ID. *Then no node broadcasts the message* "Correct-Final, V".

Proof. By Lemma 2, no prime node generates the message "Correct-Final, V". Some faulty links may create V and Correct-Final messages that contain V. Suppose that, contrary to the lemma, there are secondary nodes that broadcast the message "Correct-Final, V". Hence, there exists a node Y and a time T with the two properties that: (1) Y broadcasts at time T the message "Correct-Final, V", and (2) no node broadcasts the message "Correct-Final, V" before time T.

By the algorithm, Y must be adjacent to at least two faulty links l_1 and l_2 that deliver "Correct-Final, V" messages to Y. Suppose that, in addition to l_1 and l_2, Y is adjacent to at most $f - 2$ faulty links. In the worst case, the $f - 2$ faulty links may deliver Correct-Final and Verify-Final messages that contain V to Y.

As we explained in the general description of the algorithm, the Final-naive(Y) nodes may send "Verify-Final, V" messages to Y. In other words, the Final-naive(Y) may contribute to VecCNT(Y, V). As we discussed in the general description of the algorithm, Final-naive(Y) contains at most $t - f$ nodes.

By the algorithm, each link incident on Y contributes at most one message towards incrementing VecCNT(Y, V). Hence, VecCNT(Y, V) is at most [number of faulty links incident on Y + [number of nodes in Final-naive(Y)] $\leqq f + (t - f) = t$. Hence, the secondary node Y does not broadcast the message "Correct-Final, V", a contradiction. □

Since there are at most t faulty links in the network, and there are at least $2t + 2$ nodes in the network, there are at least two nodes P and Q that are not adjacent to any faulty links. The following lemma shows that the algorithm has the liveness property.

Lemma 4. *Let* P *and* Q *be two nodes that are not adjacent to any faulty links, and let* $\bar{V}$ *be a vector that contains the sorted* IDs *of the network nodes. Then, there exists a time after which nodes* P *and* Q *broadcast the message* "Correct-Final, $\bar{V}$".

Proof. Since P and Q are not adjacent to any faulty links, then P and Q receive copies of each Announce-ID, Verify-ID, Correct-Final, and Verify-Final message broadcast by each node. Consider the node P. The argument for Q is similar to that for P.

Suppose that, contrary to the lemma, P does not broadcast the message "Correct-Final, $\bar{V}$". By Lemma 3, and since P is not adjacent to any faulty link, P never terminates P's algorithm. Hence P continues to receive Announce-ID and Verify-ID messages from all the nodes in the network. Ergo, there will be a time T after which P certifies all the n node IDs.

Since there are t faulty links in the network, there can be at most $2t$ nodes adjacent to faulty links. Hence, there can be at most $2t$ Announce-ID messages that are lost in the network, and P will not receive the corresponding Verify-ID messages. See Fig. 3. Thus, there will be a time T* after which P receives and accepts at least $n(n-1)-2t$ Verify-ID messages that contain all the node IDs. Hence, CNT_Sum(P) after time max(T, T*) will be at least [number of Announce-ID messages in P] + [number of Verify-ID messages in P] $\geqq [n] + [n(n-1) - 2t] = n^2 - 2t$. Hence, P will compute at most t faulty links among the node IDs, and P will broadcast the message "Correct-Final, $\bar{V}$", a contradiction. □

The following lemma shows that the algorithm has the progression property.

Lemma 5. *Suppose that* $\bar{V}$ *is an* ID *vector that contains the sorted* IDs *of the network nodes. Then there exists a time after which each node* Y *broadcasts the message* "Correct-Final, $\bar{V}$".

Proof. If Y broadcasts any Correct-Final message, then, by Lemma 3, the message must contain $\bar{V}$. Hence, suppose that Y has not yet broadcast a Correct-Final message. As we explained, there are at least two nodes P and Q that are not adjacent to any faulty links. By Lemma 4, P and Q send "Correct-Final, $\bar{V}$" messages to Y and all other nodes.

Our algorithm allows each node to broadcast at most one Verify-Final message. Hence, some nodes W that receive "Correct-Final, $\bar{V}$" messages from P and Q will not broadcast "Verify-Final, $\bar{V}$" because W is a Final-naive(Y) node. As in Lemma 3, the number of Final-naive(Y) nodes is at most $t-f$. By the execution of the algorithm, the nodes that are not Final-naive(Y) nodes and that receive "Correct-Final, $\bar{V}$" messages from P and Q will broadcast "Verify-Final, $\bar{V}$" or "Correct-Final, $\bar{V}$" messages. Some of these messages may be lost in faulty links that may be adjacent to Y. Hence, VecCNT(Y, $\bar{V}$) will be at least [two messages from P and Q] + [(maximum number of messages from nodes other than P, Q, and Y) − (messages lost in faulty links) − (size of Final-naive(Y))] $\geqq [2] + [(n-3) - (f) - (t-f)] \geqq [2] + [(2t+2-3) - t] = t+1$. By the algorithm, Y broadcasts the message "Correct-Final, $\bar{V}$", and the lemma is true. □

Theorem 2. *The algorithm solves the agreement and leader election problems for asynchronous complete networks in which the processors are reliable but some channels may be Byzantine faulty. The algorithm can tolerate up to t faulty channels, provided that the total number of processors in the network is at least* $2t+2$. *When the processors terminate their corresponding algorithm, all the processors in the network will have the same correct vector, where the vector contains the private values of all the processors.*

Proof. Lemma 5 shows that there exists a time after which each node Y broadcasts a Correct-Final message that contains the vector $\bar{V}$ of the node IDs. By the algorithm, Y broadcasts the message only after Y chooses $\bar{V}$ as the final vector. □

6 Communication and storage complexity

Let S be the maximum number of bits needed to specify the node identifiers. For non-triviality, S is at least $\log_2 n$ bits.

Theorem 3. *The total number of bits that the nodes send is* $O(n^3 S)$ *bits.*

Proof. Each node Y broadcasts one Announce-ID message that contains ID(Y). Node Y broadcasts one Verify-ID message that contains ID(Y). Also, Y broadcasts one Verify-ID message for each Announce-ID message that Y accepts. Since the algorithm allows Y to accept at most one Announce-ID message per link incident on Y, Y broadcasts a total of n different Verify-ID messages. By the algorithm, Y can broadcast at most one Verify-Final message and one Correct-Final message.

Recall that each Announce-ID or Verify-ID message has one ID, while each Correct-Final or Verify-Final has n IDs. Each broadcast requires that a message be sent on each link incident on Y. Hence, the total number of bits that Y sends = (# of links incident on Y)[(# of broadcasts)(size of the messages)] $= (n-1)\ [(1)^*O(S) + (n)^*O(S) + (1)^*O(nS) + (1)^*O(nS)] = O(n^2 S)$ bits. Since there are n nodes in the network, the theorem follows. □

Theorem 4. *The amount of storage that the algorithm uses in each node is* $O(n^2 S)$ *bits.*

Proof. Each node Y use the following storage: IDMessage-List(Y, l) and VecMessage-List(Y, l) for each link l incident on Y, and CNT(Y, γ) for each message containing γ that Y accepted.

By the algorithm, IDMessage-List(Y, l) contains at most $n+1$ Announce-ID and Verify-ID messages. Hence, there can be at most $(n+1)$ CNT counters created in Y for the IDs contained in IDMessage-List(Y, l). Since Y is adjacent to exactly $n-1$ links, and Y creates a CNT counter for ID(Y), there are at most $(n+1)(n-1)+1 = n^2$ CNT counters for IDs in Y.

By the algorithm, VecMessage-List(Y, l) contains at most one message. Hence, there can be at most one VecCNT counter created in Y for the vectors contained in VecMessage-List(Y, l). Since Y is adjacent to exactly $n-1$ links, there are at most $n-1$ VecCNT counters for vectors in Y.

By Lemma 1, each CNT or VecCNT counter requires at most $\log_2 n$ bits. Hence, the total amount of storage that the algorithm uses in Y is at most (storage for IDMessage-Lists + storage for VecMessage-Lists + storage for CNT and VecCNT) $= (n-1)(n+1)^*O(S) + (n-1)(1)^*\ O(nS) + [n^2 + (n-1)]^*\log_2 n = O(n^2 S)$ bits. □

7 Conclusions

We presented the first known agreement and leader election algorithm for asynchronous complete networks in which the processors are reliable but some channels may be Byzantine faulty. When the processors terminate their corresponding algorithms, all the processors in the network will have the same correct vector, where the vector contains the private values of all the processors. In *Byzantine* failure, channels fail by altering messages, sending false information, forging messages, losing messages at will, and so on. There are no restrictions on the behavior of a faulty channel. We considered only channel failures because Fischer, Lynch, and Paterson have already shown that no algorithm can solve the agreement problem on asynchronous networks if any processor fails during the execution of the algorithm. Our asynchronous algorithm tolerates up to $\left\lfloor \frac{n-2}{2} \right\rfloor$ faulty channels, where n is the number of processors in the network. The bound on the number of faulty channels is optimal. The amount of communication and storage that our algorithm requires match the values for the Byzantine algorithms designed for *synchronous* networks.

Acknowledgements. We are indebted to the anonymous referees for their extensive comments and their help in revising the paper. In particular, we sincerely thank an anonymous referee for suggesting the proof of Lemma 2, which considerably simplified the original correctness proof and helped to shorten the paper. It is indeed a privilege to have such dedicated referees for one's papers.

References

1. Abu-Amara HH: Fault-tolerant distributed algorithm for election in complete networks. IEEE Trans Comput 37(4): 449–453 (1988)
2. Abu-Amara HH, Lokre J: Election in asynchronous complete networks with intermittent link failures. IEEE Trans Comput 43(7): 778–788 (1994)
3. Afek Y, Gafni E: Time and message bounds for election in synchronous and asynchronous complete networks. SIAM J Comput 20: 376–394 (1991)
4. Alsberg PA, Day JD: A principle for resilient sharing of distributed resources. Proc 2nd International Conference on Software Engineering, San Francisco, CA, pp 562–570, October 1976
5. Bar-Yehuda R, Kutten S: Fault-tolerant distributed majority commitment. J Algorithms 9: 569–582 (1988)
6. Ben-Or M: Another advantage of free choice: completely asynchronous agreement protocols. Proc 2nd ACM Symposium on Principles of Distributed Computing, Montreal, Quebec, Canada, pp 27–30, August 1983
7. Cristian F, Aghili H, Strong R, Dolev D: Atomic broadcasts from simple message diffusion to Byzantine agreement. Proc 15th International Symposium on Fault-Tolerant Computing, Ann Arbor, Michigan, pp 200–206, June 1985. A revised version appears as IBM Tech Rep RJ5244
8. Cimet IA, Kumar PRS: A resilient distributed protocol for network synchronization. ACM SIGCOMM Symposium on Communications, Architecture, and Protocols, Stowe, VT, August 1986, pp 358–376
9. Coan BA, Welch JL: Modular construction of a Byzantine-agreement protocol with optimal message bit complexity. Inf Computation 97(1): 61–85 (1992)
10. Dolev D: The Byzantine generals strike again. J Algorithms 3 (1): 14–30 (1982)
11. Dolev D, Dwork C, Stockmeyer L: On the minimal synchronism needed for distributed consensus. J Assoc Comput Machinery 34: 77–97 (1987)
12. Fischer MJ, Lynch NA, Merritt M: Easy impossibility proofs for distributed consensus problems. Distrib Comput 1: 26–39 (1986)
13. Fischer MJ, Lynch NA, Paterson MS: Impossibility of distributed consensus with one faulty process. J Assoc Comput Machinery 32: 374–382 (1985)
14. Frederickson GN, Lynch NA: Electing a leader in a synchronous ring. J Assoc Comput Machinery 34: 98–115 (1987)
15. Gafni E: Improvements in the complexity of two message-optimal election algorithms. Proc 4th ACM Symposium on Principles of Distributed Computing, Minacki, Ontario, Canada, August 1985, pp 175–185
16. Goldreich O, Shrira L: Electing a leader in a ring with link failures. Acta Inf 24: 79–91 (1987)
17. Korach E, Moran S, Zaks S: Tighter lower and upper bounds for some distributed algorithms for a complete networks of processors. Proc 3rd ACM Symposium on Principles of Distributed Computing, Vancouver, B.C., Canada, August 1984, pp 199–207
18. Lamport L, Shostak R, Pease M: The Byzantine generals problem. ACM Trans Program Lang Syst 4(3): 382–401 (1982)
19. Lynch N, Fischer M, Fowler R: A simple and efficient Byzantine Generals algorithm. Proc IEEE 2nd Symposium on Reliability in Distributed Software and Database Systems, Pittsburgh, PA, pp 46–52, July 1982
20. Masuzawa T, Nishikawa N, Hagihara K, Tokura N: Optimal fault-tolerant distributed algorithms for election in complete networks with a global sense of direction. Proc 3rd International Workshop on Distributed Algorithms, Nice, France, September 1989. Also in: Distributed algorithms Lect Notes Comput Sci, Vol 392, pp 171–182. Berlin Heidelberg New York: Springer, 1989
21. Mattern F: Message complexity of simple ring-based election algorithms – an empirical analysis (extended abstract). Proc IEEE 9th International Conference On Distributed Computing Systems, pp 94–100, 1989
22. Menasce DA, Popek GJ, Muntz RR: A locking protocol for resource coordination in distributed databases. ACM Trans Database Syst 5: 103–138 (1980)
23. Mohan C, Strong R, Finkelstein S: Method for distributed commit and recovery using Byzantine agreement within clusters of processors. Proc 2nd ACM Symposium on Principles of Distributed Computing, Montreal, Quebec, Canada, pp 89–103, August 1983
24. Pease M, Shostak R, Lamport L: Reaching agreement in the presence of faults. J Assoc Comput Machinery 27: 228–234 (1980)
25. Singh G: Efficient distributed algorithms for leader election in complete networks. Proc 11th IEEE International Conference on Distributed Computing Systems, pp 472–479, 1991

Appendix A: Detailed description of the algorithm

Each node Y executes the following five steps of the algorithm:

In Step 1, Y announces to all other nodes the identifier ID(Y).

In Step 2, Y parses the messages that Y receives. Depending on the type of messages, Y either executes Steps 3 and 4, or Step 5. Upon accepting an Announce-ID or Verify-ID message, Y executes Steps 3 and 4, Upon accepting a Correct-Final or Verify-Final message, Y executes Step 5.

In Steps 3 and 4, Y checks whether Y is a prime node.

In Step 5, Y checks whether Y is a secondary node.

Step 1. Each node Y sends the message "Announce-ID, ID(Y)" to all other nodes. In addition, each node Y sends the message "Verify-ID, ID(Y)" to all other nodes. Also, Y sets a counter CNT(Y, ID(Y)) with value 1, and Y labels ID(Y) as certified.

Step 2. Node Y waits until Y receives messages from at least one of the links incident on Y. Suppose that Y receives a message M on some link α. Depending on the type of message M, Y does one of the following:

(2.1) If M is an Announce-ID message, then Y extracts the ID γ contained in M and examines all the IDMessage-Lists in Y to check if Y previously accepted from any link the message "Announce-ID, γ", or if Y previously accepted from α any Announce-ID message. If the answer to either of these checks in yes, then Y discards M and executes Step 2 again. If the answer is no to both checks, then Y adds M to IDMessage-List(Y, α), labels the ID γ as certified, increments CNT(Y, γ), broadcasts the message "Verify-ID, γ" to all other nodes including the node from which Y received M, and executes Step 3.

(2.2) If M is a Verify-ID message, then Y adds M to IDMessage-List(Y, α), extracts the ID γ contained in M, increments CNT(Y, γ), and executes Step 3.

(2.3) If M is a Correct-Final message, then Y extracts the vector V contained in M and examines VecMessage-List(Y, α) to check if Y previously accepted from α any Correct-Final or Verify-Final message that contains a vector different from V. If yes, then Y discards M because M is useless, waits to receive more messages on the links incident on Y, and executes Step 2 again. If no, then Y executes the following two sub-steps:

(2.3.1) Node Y examines VecMessage-List(Y, α) to check if Y previously accepted from α the message "Verify-Final, V". Y performs this check because our algorithm requires that each link incident on Y contributes at most one message towards incrementing VecCNT(Y, V). Hence, if Y previously accepted from α the message "Verify-Final, V", then Y replaces "Verify-Final, V" in VecMessage-List(Y, α) with M, and Y executes sub-step 2.3.2 without incrementing VecCNT(Y, V). If Y did not previously accept from α the message "Verify-Final, V", then Y adds M to VecMessage-List(Y, α), increments VecCNT(Y, V), and executes sub-step 2.3.2.

(2.3.2) Node Y checks to see whether Y should broadcast the message "Verify-Final, V". Hence, Y checks if Y has already broadcast any Verify-Final message. Y performs this check because our algorithm requires that each node broadcasts at most one Verify-Final message.

If yes, the Y skips what follows and executes Step 5.

If no, then Y examines all VecMessage-Lists in Y to check if Y previously accepted a "Correct-Final, V" message from any link other than α.

If no, then Y can not yet broadcast the message "Verify-Final, V" because Y didnot yet accept two "Correct-Vector, V" on two different links. Hence, Y waits to receive more messages on the links incident on Y and executes Step 2 again.

If yes, then Y broadcasts the message "Verify-Final, V" on all links incident on Y. Node Y then executes Step 5.

(2.4) If M is a Verify-Final message, then Y extracts the vector V contained in M and examines VecMessage-List(Y, α) to check if Y previously accepted from α any Correct-Final or Verify-Final message. If the answer is yes to either of the two checks, then Y discards M because M is useless to Y, waits to receive more messages on the links incident on Y, and executes Step 2 again. If the answer is no to both checks, then Y adds M to VecMessage-List(Y, α), increments VecCNT(Y, V), and executes Step 5.

Step 3. (Y checks if Y has n certified IDs and CNT_Sum(Y) is at least $n^2 - 2t$.)

If the total number of certified IDs in Y is smaller than n, then Y waits to receive more messages on the links incident on Y and executes Step 2 again. Else, let CNT_Sum(Y) be $\sum_{i=1}^{n}$ CNT(Y, β_i), where $\beta_1, \beta_2, \ldots, \beta_n$ are the n certified IDs. Node Y checks if CNT_Sum(Y) is at least $n^2 - 2t$.

If yes, then Y executes Step 4.

If no, then Y can not be yet a prime node, and, hence, waits to receive more messages on the links incident on Y and executes Step 2 again.

Step 4. (Y checks if Y is a prime node)

For each two certified IDs γ_1 and γ_2, node Y assumes that there is a possibly faulty link between γ_1 and γ_2 if Y does not have "Verify-ID, γ_2" in IDMessage-List(Y, link(γ_1)) or if Y does not have "Verify-ID, γ_1" in IDMessage-List(Y, link(γ_2)). Next, Y checks if the number Suspect_Count(Y) of possibly faulty links computed by Y is smaller than or equal to t.

If no, then Y can not be yet a prime node, and, hence, waits to receive more messages on the links incident on Y and executes Step 2 again.

If yes, then Y is a prime node. Hence, Y sorts the IDs $\beta_1, \beta_2, \ldots, \beta_n$ in a vector V, where $\beta_1, \beta_2, \ldots, \beta_n$ are the n certified IDs in Y. Node Y sets Y's Final Vector to V, broadcasts the message "Correct-Final, V" to all other nodes, and *stops executing the algorithm.*

Step 5. (Y checks if Y is a secondary node)

Node Y checks if VecCNT(Y, V) is at least $t + 1$ and if there exists at least two VecMessage-Lists in Y that contain "Correct-Vector, V".

If the answer is no to either of the two checks, then Y can not be yet a secondary node, and, hence, waits to receive more messages on the links incident on Y and executes Step 2 again.

If the answer is yes to both checks, then Y is a secondary node. Node Y sets Y's Final Vector to V, broadcasts the message "Correct-Final, V" to all other nodes, and *stops executing the algorithm.*

Hasan M. Sayeed received his B.S. (December 1989) degree in electrical engineering from the Bangladesh University of Engineering and Technology, Bangladesh, and the M.S. (August 1992) degree in electrical engineering from Texas A&M University, College Station, Texas. He is a Ph.D. student in the Department of Electrical Engineering at Texas A&M University. Hasan M. Sayeed's research interest are in distributed computing, secure and secret communication in computer networks, and fault-tolerant computer networks.

Marwan Abu-Amara received the B.S. (May 1988) degree in computer engineering from Kuwait University, Kuwait, and the M.S. (December 1991) and Ph.D. (May 1995) degrees in electrical engineering from Texas A&M University, College Station, Texas. He is a member of the scientific staff at Bell-Northern Research in Richardson, Texas. His research interests are in wireless communication, distributed computing, computer networks, microprocessor-based system design, and fault-tolerant and testable design.

Hosame Abu-Amara received the B.S. (March 1983) degree in electrical engineering from the University of California, Berkeley, and the M.S. (May 1985) and Ph.D. (October 1988) degrees in electrical engineering from the University of Illinois at Urbana-Champaign. He is an Assistant Professor with the Department of Electrical and Computer Engineering at the University of Nevada in Las Vegas. His research interests are in fault-tolerant distributed computing, secure and secret communication in computer networks, fiber-optic networks, and implementation of communication protocols in hardware.