PUC Minas
A B A K Ó S
Instituto de Ciências Exatas e Informática

Alternative specification and correctness proofs of the distributed network reachability algorithm*

Especificação e prova de correção alternativas para o algoritmo DNR de diagnóstico de redes de tepologia arbitrária

Andréa Weber[1]
Elias P. Duarte Jr.[2]
Keiko V. O. Fonseca[3]

Resumo

O algoritmo *Distributed Network Reachability* (DUARTE-JR; WEBER; FONSECA, 2012) permite que cada nodo em uma rede de topologia arbitrária determine quais porções da rede estão alcançáveis e inalcançáveis. O algoritmo consiste de três fases: teste, disseminação de informação sobre novos eventos e cálculo de alcançabilidade. Durante a fase de testes, cada enlace é testado por um de seus nodos adjacentes em intervalos de teste alternados. Quando da detecção de um novo evento, o testador inicia a fase de disseminação. Neste trabalho uma especificação alternativa do algoritmo DNR é apresentada, que emprega *tokens* na fase de testes. O trabalho também inclui todo um conjunto alternativo de provas de correção do algoritmo.

Palavras-chave: Alcançabilidade da rede. Diagnóstico distribuído. Assinalamento de testes baseado em token. Sistemas particionáveis. Recuperação de enlaces.

*Invited paper
[1]Dept. Informatics, Federal University of Paraná, Curitiba - PR, Brazil – andrea@inf.ufpr.br.
[2]Dept. Informatics, Federal University of Paraná, Curitiba - PR, Brazil – elias@inf.ufpr.br.
[3]Graduate Program on Electrical Engineering and Industrial Informatics, Federal University of Technology, Curitiba - PR, Brazil – keiko@utfpr.edu.br.

Abstract

The *Distributed Network Reachability* algorithm (DUARTE-JR; WEBER; FONSECA, 2012) allows every node in a general topology network to determine which portions of the network are reachable and unreachable. The algorithm consists of three phases: test, dissemination, and reachability computation. During the testing phase each link is tested by one of the adjacent nodes at alternating testing intervals. Upon the detection of a new event, the tester starts the dissemination phase. In this work we both give an alternative specification of DNR that employs tokens at the testing phase allowing the pair of nodes connected by a link to share testing responsibilities, and give an alternative set of proofs for the algorithm.

Keywords: Network reachability. Distributed diagnosis. Token-based testing assignment. Partitionable systems. Healing networks.

1 Introduction

Organizations and individuals increasingly depend on the correct behavior of network-based systems. In order to improve the dependability of any computer system, the first step is to have a dependable monitoring strategy. In particular, considering networks of arbitrary topology, it is important to be able to determine at any instant of time which portions of the network are reachable and which portions are unreachable. In this work we present an alternative specification and set of proofs of the Distributed Network Reachability (DNR) algorithm. The algorithm was originally published at (DUARTE-JR; WEBER; FONSECA, 2012) and presents a fault-tolerant approach for on-line network monitoring.

The alternative specification and correctness proofs presented in this paper are important because as they give a different perspective on the algorithm, they allow a comprehensive understanding of its main principles. The set of proofs presented in (DUARTE-JR; WEBER; FONSECA, 2012) is based on the theoretical framework known as Bounded Correctness (SUBBIAH; BLOUGH, 2004). In this paper we employ a set of proofs which does not rely on that framework, and may be more intuitive for readers which are not familiar with it.

The system model employed assumes a general topology network in which there is not necessarily a communication channel between any pair of nodes. Both nodes and links may be either faulty or fault-free at a given instant of time. We consider both crash and timing faults; a fault may partition the network. Considering a pair of nodes connected by a link, if the tests executed on one node by the other determines that the link is unresponsive, it is impossible to determine whether the tested node or the link is faulty. In this way, faults in a general topology network are said to be ambiguous (DUARTE-JR. et al., 1997). In the proposed algorithm, a node may assume one of two states: *working* or *perceived unreachable*. A timeout on a test executed on a given node corresponds to a perceived unresponsive state that is recorded as an event. A link may be either *working*, *unresponsive* or *perceived unreachable*. A node considers a link to be perceived unreachable when the link is not adjacent to any other reachable fault-free node.

System-level diagnosis (MASSON; BLOUGH; SULLIVAN, 1996) of general topology networks was first proposed by Bagchi and Hakimi in (BAGCHI; HAKIMI, 2002). Their algorithm cannot be used for continuous network connectivity monitoring because it is executed off-line. In (STAHL; BUSKENS; BIANCHINI, 1992) Bianchini and Buskens introduced and evaluated through simulation the Adapt algorithm, which can be executed on-line: when a given node becomes faulty, a new phase begins in which other nodes reconnect the testing graph. The algorithm employs a distributed procedure that employs a sequential event dissemination strategy.

Rangarajan, Dahbura and Ziegler (RANGARAJAN; DAHBURA; ZIEGLER, 1995) introduced the RDZ algorithm for system-level diagnosis for networks of arbitrary topology that can be executed on-line. The algorithm builds a testing graph that guarantees the optimal number of tests, i.e., each node has one tester. Furthermore it presents the best possible diagnosis latency by using a parallel dissemination strategy. Whenever a node detects an event, it sends diagnostic information to all its neighbors, which in turn send the information to all its neighbors, and so on. Although the RDZ algorithm presents the best possible diagnosis latency, and the best possible number of testers per node, it only guarantees the eventual diagnosis of the so called jellyfish fault configuration.

A diagnosis algorithm for non-broadcast network diagnosis was introduced in (DUARTE-JR. et al., 1997). The algorithm is based on a token-based testing strategy which was later modified (SIQUEIRA; FABRIS; DUARTE-JR, 2003) so that nodes connected by a link alternate in the roles of tester and tested node. The algorithm assumes a static fault situation, i.e. a new event only occurs after the previous event has been completely diagnosed. Partitions are not allowed. A diagnosis algorithm for computing network connectivity was introduced in (DUARTE-JR.; WEBER, 2003) which supports dynamic events, i.e. during the dissemination phase new events may occur and the diagnosis of all events is guaranteed within a given latency. Later another testing strategy was developed (WEBER; DUARTE-JR.; FONSECA, 2003) that deals with situations in which adjacent nodes are repaired and start testing each other simultaneously. These algorithms do not support network partitions.

All the algorithms above were proven to be correct under a static fault situation, in which a node or link is assumed to change state only after the diagnosis of a previous event completes. Subbiah and Blough introduced in (SUBBIAH; BLOUGH, 2004) a formal model of the dynamic behavior of diagnosis algorithms, called Bounded Correctness, which allows a diagnosis algorithm to be rigorously proven to be correct under a dynamic fault situation. Bounded Correctness has three goals. The first goal is to show how quickly a working node learns about every event in the system. The second goal is to show that the views that recovering nodes obtain about the state of other nodes are out-of-date by only a bounded amount. Finally the third goal is to show that working nodes do not detect any spurious events. In the same work, the authors introduce algorithm ForwardHeartbeat, which allows the diagnosis of general topology networks in a dynamic fault situation. The algorithm implements tests implicitly by having every fault-free node send heartbeat messages at predefined time intervals. ForwardHeartbeat does not allow network partitions.

The Distributed Network Reachability (DNR) algorithm for which we describe a specification alternative to the one presented in (DUARTE-JR; WEBER; FONSECA, 2012) employs a testing strategy that guarantees the optimal number of tests per testing interval, as long as the clock of any node does not run twice or more as fast as its neighbors' clocks. Upon the detection of a new event, i.e. the change of the perceived state of a tested link, a message carrying new diagnostic information is disseminated to the other reachable nodes. New events can occur at any time during the execution of the algorithm. Whenever an event is detected or informed, a graph connectivity algorithm is employed to compute the network reachability from the perspective of any node. The specification presented in this paper is based on the idea that nodes exchange a token, so that the node which has the token is the tester for the corresponding link in the present testing interval.

The rest of the paper is organized as follows. In section 2 the system model and the alternative algorithm specification are given. In section 3 we prove the correctness of the testing phase. Section 4 presents proofs of correctness of the dissemination phase. Section 5 concludes the paper.

2 The Distributed Network Reachability Algorithm

In this section, the system model is given and the alternative specification of the DNR algorithm is presented.

2.1 Diagnosis and System Model

Consider a network or system of arbitrary topology, which is represented by a graph $S = (V(S), E(S))$ where $V(S)$ is a set of N vertices or nodes, n_0, n_1,...,n_{N-1}, and $E(S)$ is the set of undirected edges or links. We alternatively refer to node n_i as *node* i. Each edge that connects node i and node j is represented by the pair (i, j). Node i and node j are then said to be *neighbors*.

The system is synchronous, but timing faults are also possible, in the sense that a node may be slower than expected to execute tasks or send messages. Both nodes and links can become faulty by crashing. A crashed link corresponds for instance to a severed physical link. Working links are assumed to offer a reliable communication service. A fault may partition the network, which can later heal as faulty units recover. Nodes are diagnosed as *working* or *perceived unreachable*, and links are diagnosed in one of three states: *working*, *unresponsive* or *perceived unreachable*.

An *event* is defined as a change of the state of a system unit. An event is *always* recorded as a link state change, even if it occurred at a node. The link state change is either from *working* to *unresponsive* or from *unresponsive* to *working*.

2.2 Algorithm Description

The Distributed Network Reachability (DNR) algorithm is executed on-line continuously: nodes periodically execute tests on adjacent links. Upon the detection of a new event on an adjacent link, the tester starts the dissemination of new event information to reachable working nodes. At any time a node can run a graph connectivity algorithm locally to compute the network reachability.

A test assigment for system S at a given testing interval is represented by a directed graph, $T(S) = (V(S), A(S))$, where the set of vertices corresponds to network nodes, and $A(S)$ is the set of arcs correponding to the tests executed. An arc (i, j) directed from node i to node j corresponds to a test executed by node i on node j. DNR employs a dynamic testing assignment, in the sense that two nodes connected by a communication link alternate in the roles of *tester* and *tested*. The link connecting those nodes is called the *tested link*. Tests are assumed to consist of a complete procedure tailored to the system technology. Tests are executed periodically at a testing interval.

Nodes keep a local view of the network topology represented as a graph, in which a *timestamp* is kept for each link. Timestamps are event counters, similar to those employed in (RANGARAJAN; DAHBURA; ZIEGLER, 1995). The timestamps for all links are initially set to 1, and all links are assumed to be *unresponsive*. Every time a new event is detected, the timestamp for that link is incremented. Thus an even timestamp corresponds to a *working* link; an odd timestamp corresponds to an *unresponsive* link.

After a node starts-up or is repaired, it waits a pre-defined time interval called *node recovery wait time* (SUBBIAH; BLOUGH, 2004) before sending or responding either test or dissemination messages. This interval guarantees that unresponsive links are always detected, even if the node quickly recovers. At start-up, after the node recovery wait time elapses, a node running the algorithm tests all its neighbors. As a tested node receives a test request it also determines the tester is working; this strategy is called a *two-way test* (DUARTE-JR. et al., 1997).

DNR employs a token-based test assignment based on two-way tests. Each pair of nodes connected by a link keeps one token. The node that has the token at the start of a given testing interval is the tester. If the test is successful, the token is transferred, inverting the nodes' roles. As long as no events occur, the process is repeated, guaranteeing that only one test is executed per link per testing interval.

Even in case the link is tested as *unresponsive*, i. e. the tester times out waiting for a reply, the token is released. In order to guarantee that the detected unresponsiveness is correctly diagnosed, even if it corresponds to a timing fault, the tester does not communicate on the link for a pre-defined time interval called *link recovery wait time*. In one more interval, after the previous tester has assumed the role of tested node and detects it has not been tested, the token is recreated. The node then becomes the tester again, and executes a test at the next interval. As long as no new events are detected, a test is executed each two testing intervals. On the other hand, if the tester for the next testing interval becomes faulty, while the tested node remains fault-free, the token disappears. The proposed strategy leads the tested node to detect that it wasn't tested and in the next testing interval a token is created and the node assumes the role of tester, also executing a test each two intervals. As soon as a faulty node recovers it creates a token and starts up as described above.

In case both nodes become faulty, no test is executed on the link. If both nodes are repaired at roughly the same time and become testers, a notable situation arises: both create tokens and may execute *simultaneous tests*. A similar situation arises when a faulty node recovers and tests a neighbor at the same time it is tested by that neighbor. The algorithm employs a strategy in order to prevent that both nodes become testers at the same interval. As both receive a test request, one from each other, only the node with the smaller identifier will reply to the test, thus becoming the tested node.

The testing strategy also works correctly if the link is faulty or messages arrive later than expected, while both connected nodes are working. In this case both nodes will eventually create tokens and each will run a test once every two testing intervals. After the link recovers, the first test issued by one of the nodes causes the other to reply and become the tester for the subsequent interval. Simultaneous tests may also occur and are solved as described above.

Upon the detection of a new event on an adjacent link, the tester starts the dissemination of a message with the new information. An event corresponds to an adjacent *working* link becoming *unresponsive* or vice versa. The dissemination message contains information about new events. The information consists of (1) the tester identifier, (2) the tested node identifier, and (3) the corresponding timestamp. The algorithm employs a simple parallel dissemination strategy with latency proportional to the network diameter. The node that starts the dissemination sends the message on all its working adjacent links. Dissemination messages are acknowledged. A node that receives a dissemination message verifies whether the message contains new information or not. Information is new when a received timestamp is greater than the corresponding timestamp stored in the local link state table. Old information is discarded. If the message contains new information, it continues to be disseminated. The node forwards the message on all adjacent links, except the link(s) from which the message arrived.

After a dissemination is started and before it completes, new events may occur. Furthermore, a node may detect new events during the dissemination, e.g. it may send a message on a link considered to be *working* and detect the link has become *unresponsive*. If two or more disseminations start concurrently at different nodes of the same connected component, they may arrive simultaneously at some node, from which they proceed and complete independently.

Notably, a node or link fault that partitions the network confines the dissemination to one or more connected components. As opposed to it, the occurrence and detection of a healing event may have the opposite effect: the message will reach previously unreachable portions of the network. When a working node receives information about an *unresponsive* link, it computes the network reachability and sets the timestamps of every *unreachable* link to 1, the smallest possible timestamp.

A *healing event* is defined by an *unresponsive* link being detected as *working*. This may or may not cause two or more partitions to merge in one connected component. If partitions are merged, nodes in each partition must take into account the events that occurred at previously unreachable units. A healing event is detected as a test request arrives from a link considered to be *unresponsive*. In this case a *healing message* is employed, which carries information about link state table entries with timestamps greater than 1. Upon the reception of a healing message, the tester first updates its local link state table with new information, then increments the timestamp of the healed link and starts the dissemination of full link state table information to all its neighbors. Only new information is further disseminated. This strategy allows both nodes to exchange information about previously unreachable components.

A particular case happens if a node crashes and recovers several times while its neighbors remain working, then those nodes successively increment the timestamps of the links adjacent to the node. When that node recovers, it must receive link state table information from its neighbors before incrementing the timestamps of its healing links. Another special case occurs when a node becomes faulty just after sending a test request to a recovering node. If the recovering node then assumes the role of tested for that link, due to simultaneous tests, its test reply arrives at the tester when that node is faulty and the healing does not complete. Thus the timestamp for that link must not be incremented.

2.3 Algorithm Specification

The specification presented here is an alternative version of the specification present in (DUARTE-JR; WEBER; FONSECA, 2012). This specification describes the alternance of the roles of the nodes connected by a link as testers and tested based on the exchange of a *token*.

The testing strategy is given in pseudo-code in figures 1, 2 and 3. Figure 1 shows the local data structures kept by a node running DNR, and the initialization procedure. An `Event` is described by a link identifier `LinkId`, which consists of the identifiers of the tester and the tested nodes, and by the corresponding `Timestamp`. `LinkStateTable` is the data structure that keeps the local view of the topology. `Token[j]` and `TokenTurn[j]` are variables used to control the token exchange. `TestRequestSent[j]` and `TestRequestReceived[j]` are used to detect simultaneous tests. The `TestingInterval[j]` timer is used to define the period of time a test is executed; the `TestTimeout[j]` timer defines the corresponding timeouts employed to conclude link unresponsiveness.

As shown in Figure 1, when the node is first initialized or recovers, the `NodeRecoveryWaitTime` timer is started. The node delays sending test requests until the timer expires. As `Token[j]` is set to TRUE at the initialization of the algorithm, the node is forced to test all its neighbors after the *node recovery wait time*.

Figure 1 – DNR Algorithm: Initialization and Startup.

Distributed Network Reachability Algorithm Executed by Node i

```
define NodeId = node's network address;
       Timestamp = counter;
       LinkId = (NodeId, NodeId);
       Event = (LinkId, Timestamp);
       Message = list of Event;

var LinkStateTable: array[LinkId] of Timestamps;
    Token, TokenTurn: array[NodeId] of Boolean;
    TestRequestSent, TestRequestReceived: array[NodeId] of Boolean;
    TestingInterval, TestTimeout: array[NodeId] of Timers;
    NodeRecoveryWaitTime, LinkRecoveryWaitTime: Timer;

Start&RunForever()
for all links do LinkStateTable[LinkId] = 1; end for
for each neighbor j
    Token[j] = TRUE;
    TokenTurn[j] = FALSE;
    TestRequestSent[j] = FALSE;
    TestRequestReceived[j] = FALSE;
    start_timer NodeRecoveryWaitTime;
end for
when NodeRecoveryWaitTime timer expires
  for each neighbor j
      expire_timer TestingInterval[j];
  end for
end when
while TRUE do
  for each neighbor j
      when TestingInterval[j] timer expires
            restart_timer TestingInterval[j];
            TokenTest(j)
      end when
  end for
end while
```

In figure 2 module `TokenTest(j)` shows the strategy for alternating the token. If the node running the algorithm has the token and is supposed to test a neighbor *j* in the next testing interval, the `RunTest(j)` module is executed and the token is released as `Token[j]` is set to FALSE. Otherwise, if a link turns out to be unresponsive, i.e., `Token[j]` is set to FALSE and `TokenTurn[j]` is also set to FALSE, then `TokenTurn[j]` is set to TRUE to cause a test to be executed in the next testing interval. In case `Token[j]` is FALSE but `TokenTurn[j]` is TRUE, a new token is created and `TokenTest(j)` is called.

Figure 2 – DNR Algorithm: Testing strategy as executed by a tester.

Distributed Network Reachability Algorithm Executed by Node i

```
|| Module: TokenTest(j)
  if Token[j]
  then Token[j] = FALSE;
       RunTest(j);
  else if TokenTurn[j]
       then TokenTurn[j] = FALSE;
            Token[j] = TRUE;
            TokenTest(j);
       else TokenTurn[j] = TRUE;
       end if
  end if

|| Module: RunTest(j)
  if (LinkStateTable[i,j] mod 2 != TestRequest(j)) /* new event detected */
  then if (LinkStateTable[i,j] mod 2 != 0) /* healing event */
       then update LinkStateTable with new info;
            LinkStateTable[i,j]++;
            Event-Disseminate(LinkStateTable); /* send timestamps > 1 */
       else LinkStateTable[i,j]++;
            Event-Disseminate(msg with new info);
            during LinkRecoveryWaitTime ignore msgs to/from link(i, j)
       end if
       run local graph connectivity algorithm;
       for all unreachable links do LinkStateTable[LinkId] = 1; end for;
  end if

|| Module: TestRequest(j)
  send(j, test request);
  TestRequestSent[j] = TRUE;
  TestRequestReceived[j] = FALSE;
  start_timer TestTimeout[j];
  when the test reply from j arrives
       reset_timer TestTimeout[j];
       return 0; /* working */
  end when
  when TestTimeout[j] expires
       if TestRequestReceived[j]
       then Token[j] = TRUE;
       end if
       return 1; /* unresponsive */
  end when
```

`RunTest(j)` is called by the node that has the token and executes the test procedure. If the link turns out to be unresponsive, the dissemination of a message containing information about the event is started. Furthermore, during the interval specified by timer `LinkRecoveryWaitTime`, node *i* ignores all messages to/from link (*i, j*) in order to guarantee that the detected unresponsiveness is correctly diagnosed, even if it corresponds to a timing fault. In case of a healing event, the full link state table, updated with information received from the tested node, is disseminated. A graph connectivity algorithm is run to compute the network reachability. Timestamps of *unreachable* links are set to 1 in order not to be disseminated as spurious events in case of a healing.

`TestRequest(j)` describes the testing procedure. The `TestTimeout[j]` timer is initialized and a test request is sent to the tested node *j*. Variables `TestRequestSent[j]` and `TestRequestReceived[j]` are set each time a test request is sent, in order to allow the detection of simultaneous tests, as shown below.

In figure 3, module `TestReply(j)` corresponds to the procedure run by the tested node as it replies to a test request. The token is exchanged, as `Token[j]` is set to TRUE. Furthermore, as tests are *two-way*, when a test request arrives from a link considered to be *unresponsive*, the tested node detects a healing event on that link. The `LinkStateTable` of the tested node is then sent to the tester as a reply. Finaly, the `TestingInterval[j]` timer is restarted.

Figure 3 – DNR Algorithm: Testing strategy as executed by the tested node.

Distributed Network Reachability Algorithm Executed by Node i

```
|| Module: TestReply(j)
  when a test request from j arrives
      Token[j] = TRUE;
      TokenTurn[j] = FALSE;
      if TestRequestSent[j] /* simultaneous testing */
      then TestRequestSent[j] = FALSE;
            if $i > j$
            then Token[j] = FALSE; /* it's i's turn to test */
                  TestRequestReceived[j] = TRUE;
                  return;
            else reset_timer TestTimeout[j];
            end if
      end if
      restart_timer TestingInterval[j];
      if (LinkStateTable[i,j] mod 2 != 0) /* healing event detected */
      then send(j, test reply with LinkStateTable); /* send timestamps $> 1$ */
      else send(j, test reply);
      end if
  end when
```

Variable `TestRequestSent[j]` is checked to determine whether simultaneous tests have occurred. Node i running DNR detects simultaneous tests when a test request arrives from node j and the local `TestRequestSent[j]` is set to TRUE. A criterion based on the identifier of the nodes is used to solve the problem, and guarantees that only one node is the tester for the next interval. The node with the smaller identifier sends a test reply and resets the corresponding `TestTimeout` entry.

In a very particular case, the test request sent by node i arrives at node j while node j is still in the *node recovery wait time*. In this case if node j completes the *node recovery* *wait time* and sends a test request to i before i times out on the expected reply, simultaneous tests are detected by node i but not by node j. If node i has the largest identifier, it would become the tester if simultaneous tests were detected, but both nodes time out waiting for the test replies. In order to avoid this situation, the node with the largest identifier sets the corresponding `TestRequestReceived` entry to TRUE upon detecting simultaneous tests. If that node times out, it resets the corresponding `Token` entry to TRUE in order to test again in the next testing interval. Then the healing event will be detected. This is the only situation in which the same node tests in two consecutive testing intervals.

The specification of the dissemination strategy is shown in figure 4. A `Message` consists of a list of `Events`. No message identifier is required, because the list of events is enough to make each message unique. `LinkStateTable` is the data structure used to keep the local view of the topology, `Timestamps` are initially set to 1, and are updated with every event detected on an adjacent link or informed with a dissemination message.

Figure 4 – DNR Algorithm: Dissemination of new event information.

Distributed Network Reachability Algorithm Executed by Node i

```
|| Module: Event-Disseminate(msg)
  for each node j neighbor of node i
      if (j != msg's senders)
      then if ((LinkStateTable[i,j] mod 2 == 0) and (send(j, msg) does not succeed)) /* fault event detected */
           then LinkStateTable[i,j]++;
                Event-Disseminate(msg with new info);
                during LinkRecoveryWaitTime ignore msgs to/from link(i, j)
                run local graph connectivity algorithm;
                for all unreachable links do LinkStateTable[LinkId] = 1; end for
           end if
      end if
  end for

|| Module: Receive(msg)
  when a dissemination message from j arrives
      if (msg has new information)
      then Event-Disseminate(msg with new info);
           update LinkStateTable with new info;
           run local graph connectivity algorithm;
           for all unreachable links do LinkStateTable[LinkId] = 1; end for
      end if
  end when
```

The dissemination is started when a new event is detected (Module `RunTest(j)`). `Event-Disseminate(msg)` is called and receives as parameter the message to be disseminated. In case of a fault event, this event alone is disseminated. In the case of a *healing* event, a message is sent to neighbors with an extract of the local view of the topology kept in `LinkStateTable`.

`Event-Disseminate(msg)` is executed to send the message to each neighbor. Nevertheless, as the message is sent, a fault event may be detected that triggers a new dissemination. If this occurs, during an interval equal to the `LinkRecoveryWaitTime` node *i* ignores all messages to/from link (*i, j*). As is always the case when a new event is detected, the local connectivity algorithm is run, allowing the timestamps of unreachable links to be reset to 1.

`Receive(msg)` employs the criterion based on timestamps to determine if a received message carries new information. Old information, if any, is discarded. After `Event-Disseminate(msg)` is executed, the local `LinkStateTable` is updated. The message is forwarded on all links, except the one(s) from which it arrived. The local graph connectivity algorithm is run.

3 Testing Phase Proofs

In this section we prove both the correctness of the testing phase and its worst-case detection latency. We show that the algorithm is optimal: even if two nodes start up or recover and test each other simultaneously, the algorithm guarantees that from the next interval on the nodes alternate testing their adjacent link in successive testing intervals. If both nodes connected through a working link are also working, then the algorithm guarantees that only one test is executed per link per testing interval.

The proof is organized as follows. First we show that after a node starts up it tests every adjacent link once every two testing intervals. Then we prove that after a new event occurs on a link adjacent to a *working* node, the node continues testing the link once every two testing intervals. Then we use this for prooving the worst-case event detection latency of the algorithm.

DEFINITION 1. An *unresponsive* link adjacent to a *working* node is said to *heal* if either the neighbor starts up, while the link remains *working*, or both nodes remain *working* while their adjacent *unresponsive* link recovers. The corresponding event is called a *healing event*.

The testing algorithm is based on a control message called a *token* that is *exchanged* by two working nodes through a *working* communication link in successive testing intervals. The *testing interval* is a time interval after which a given node that has a token executes a test. Nodes are not synchronized with each other, and do not share a global clock. The tester restarts its testing interval after sending a test request. The tested node also restarts its testing interval upon receiving the test request.

As soon as a *working* node running DNR sends a test request, it *releases* the token and becomes the tested node for the next testing interval. Likewise, as soon as a *working* node running DNR receives a test request, it replies and *obtains* the token, becoming the tester for the next testing interval. The tester also releases the token even if it sends a test request and the tested node does not reply, i.e. if the adjacent link is *unresponsive*. A new token is *created* if none is received in two consecutive testing intervals.

We assume that the difference of the clock speeds of any two neighbors is less than one the double of the other. We later show what happens when one of the clocks is twice or more faster than the other.

DEFINITION 2. Two nodes are said to be *simultaneous testers* when both are connected by a *working* link and have tokens in the same testing interval.

DEFINITION 3. Two tests are said to be *simultaneous* when two simultaneous testers send test requests to each other before each receives a test request from the other.

It is important to note that simultaneous testers do not execute simultaneous tests if the test request from one of them happens to reach the other before its test request is sent.

DEFINITION 4. A node is said to *start up* when it sends a round of test requests after the *recovery wait time*. When simultaneous tests occur, another round of test requests may be necessary, as will be shown below. After start-up, a node running DNR tests an adjacent link once every two testing intervals.

LEMMA 1. After a node running DNR starts up and tests an adjacent link as *unresponsive*, the node continues testing that link once each two testing intervals, as long as the link remains *unresponsive*.

PROOF. According to the specification of the algorithm, once a node starts up or recovers it creates a token and tests all its neighbors. After testing an adjacent link, the node releases the corresponding token, and becomes the tested node in the next testing interval. If the link to that neighbor remains *unresponsive*, the node does not receive a test request and a token is not exchanged. The node then sets variable `TokenTurn` meaning that a token must be created in the following testing interval. Thus, in the following testing interval a token is created and the node tests the link and releases the token again. From this interval on the same pattern is repeated, as long as the link remains *unresponsive*. Thus, the node keeps testing the link once each two testing intervals. □

The following lemma shows the situations in which simultaneous testers may occur.

LEMMA 2. When a node starts up and an adjacent link is *working*, if the corresponding neighbor is either starting up or is already *working*, these two nodes may become simultaneous testers. Simultaneous testers may also occur if both nodes are *working* while their adjacent *unresponsive* link recovers. Simultaneous testers also occur if two *working* nodes are connected by a *working* link and the clock of one of them is twice faster than the clock of the other.

PROOF. According to the specification of the algorithm, after a node starts up it creates a token and becomes a tester for that interval. If a neighbor is also starting-up, it also creates a token, thus both become simultaneous testers for their adjacent link.

Now if a node starts up while a neighbor is already *working*, this *working* neighbor is testing the corresponding link once each two testing intervals, as stated by LEMMA 1. As the initializing node creates a token the *working* neighbor may also be a tester in that testing interval.

If both nodes connected by an *unresponsive* link are *working*, then both are testing that link once each two testing intervals, as stated by LEMMA 1. As they are not exchanging their tokens, both may be testers after their adjacent link recovers.

Finally, consider the case in which the clock of a *working* node is twice as fast as the clock of a *working* neighbor, and they are connected by a *working* communication link. In this case, as soon as the faster node sends a test request it restarts its testing interval, and becomes the tested node. The new tester, i.e., the node that receives the test request, also restarts its testing interval. Since the testing interval of this node is twice slower than the testing interval of the other node, the faster node becomes a tested node and again a tester while the slower node is still also a tester. Thus, periodically both nodes become simultaneous testers. □

THEOREM 1. When two nodes are simultaneous testers, only one of them keeps the token for the next testing interval.

PROOF. First consider that both nodes are simultanous testers but before one node sends a test request, it receives a test request, i.e. simultaneous tests do not occur. The node replies to the test, and does not test the link in that interval.

Now consider that simultaneous tests occur. Accoding to the specification of the algorithm, as a node executes a test, it sets variable `TestRequestSent`, meaning that, for that link, a test request was sent. If two nodes execute simultaneous tests then each will have its variable `TestRequestSent` set for the same link. As both nodes receive test requests, both check that variable and realize that a simultaneous test has occurred, and only the node with the smaller identifier replies, thus becoming the tested node. The other node does not reply to the test, and behaves as the only tester in that testing interval. In the next testing interval only the current tested node will have the token.

A notable situation occurs when two nodes connected by a *working* link are simultaneous testers and send test requests, but the test request of one of them arrives at the other when that node is still in *recovery wait time*. Call the node that sent this test request node *A* and node *B* the other node. According to DEFINITION 3, the tests are not simultaneos, once the test of node *A* arrives at node *B before* that node sends its test request. Nevertheless, if the test request sent by node *B* at the end of *recovery wait time* arrives at node *A* before that node times-out, then, according to the algorithm specification, node *A* detects the occurrence of simultaneous tests because that node has previously set variable `TestRequestSent[j]`.

At this point, one of two situations may arise. If node *A* has the largest identifier, it becomes the tester node and sets variable `TestRequestReceived[j]`. Because its test request is dropped, node *A* times-out and sets variable `Token[j]`, in order to test again at the next testing interval. In this case, node *B* does not receive a reply for its test request. As this node did not receive a test request, it also does not detect simultaneity of tests. According to the specification of the algorithm, it becomes the tested node for the next testing interval. Thus, only one tester remains.

If, on the other hand, node *A* has the smallest identifier, it assumes the role of tested node when detects simultaneity of tests. So, it sends a reply and gets the token, to be the next tester. Thus, in each of the situations (the node that detected simultaneity of tests, i.e., node *A*, becomes the tester or the tested node), only that node has the token for the next testing interval.

Thus, when two nodes are simultaneous testers, only one of them has the token for the next testing interval. □

LEMMA 3. After a node starts up and tests an adjacent link as *working*, the node tests that link once each two testing intervals as long as the link and the nodes remain *working*.

PROOF. A node that starts up tests an adjacent link as *working* either if the neighbor is starting up or *working* and also the link is *working*. According to LEMMA 2, in these situations both nodes may become simultaneous testers. Theorem 1 assures that only one node keeps the token for the next testing interval, either if simultaneous testers execute simultaneous tests or not. Furthermore, if the nodes are not simultaneous testers, the node that has just started up or recovered sends a test request to the other successfully.

In both cases (nodes become simultaneous testers or not), the node that sends the test request also releases the token. The node that receives the test request sends a test reply and gets the token, becoming the tester for the next testing interval. From that interval on they keep alternating their roles of tester and tested as long as both nodes and the link remain *working*. Thus the node that has just started up tests that link once each two testing intervals. □

LEMMA 4. If a *working* node is adjacent to a link that becomes and remains *unresponsive*, the node tests that link once each two testing intervals.

PROOF. A node running DNR either tests or is tested on a given link in a given interval. If a link becomes *unresponsive* and the *working* node is the tested node in the next testing interval it will not receive a test request in that interval. Thus, in the next testing interval the node sets variable `TokenTurn` so that a token is created in the following interval. After the corresponding test is executed, the node releases the token, thus becoming the tested node for the following testing interval as in the initial situation, and this way successively. Now if the node is the tester after the link becomes *unresponsive*, it releases the token as soon as it executes the test. From that testing interval on, the same pattern as described above is repeated, for the node becomes the tested node and the link remains *unresponsive*. Thus the node keeps testing the link once each two testing intervals. □

LEMMA 5. If a node is *working* and a healing event occurs on an adjacent link, this node keeps testing that link once each two testing intervals as long as the adjacent link and the nodes remain *working*.

PROOF. A healing event occurs either because a neighbor starts up or recovers, or a link recovers while both adjacent nodes are *working*. According to LEMMA 2 when a healing event occurs both nodes may become simultaneous testers, and simultaneous tests may occur. Theorem 1 guarantees that only one node succeeds as the tester. If the nodes do not become simultaneous testers, then either the node or the neighbor send a test request to the other node sucessfully.

In both cases (nodes become simultaneous testers or not), the node that sends the test request also releases the token. The node that receives the test request sends a test reply and gets the token, becoming the tester for the next testing interval. From that interval on they keep alternating their roles of tester and tested as long as both nodes and the link remain *working*. Thus the node that was *working* and detected the healing event tests that link once each two testing intervals. □

THEOREM 2. If the clock of a *working* node is more than twice as fast as the clock of a *working* neighbor, and they are connected by a *working* communication link, then the only tester of the link is the faster node, and it tests the link once every two testing intervals.

PROOF. As soon as the faster node tests the link, the neighbor replies and restarts its testing interval. The neighbor is thus supposed to be the next tester. Nevertheless the testing interval of the faster node expires twice before the time instant in which the neighbor is supposed to execute a test. Thus the next test is again executed by the faster node.

In other words: after its testing interval expires for the first time, the faster node sets variable `TokenTurn`. When the testing interval expires again, the faster node executes a test, and this is before the neighbor's testing interval has expired. After this new test is executed, the other node restarts its testing interval again, and this way successively.

Thus, in this case, only the faster node tests the link every two testing intervals. □

THEOREM 3. If the clock of a *working* node is exactly twice as fast as the clock of a *working* neighbor, and they are connected by a *working* communication link, then either they alternate as tester and tested nodes or only one of them tests the link, depending on their identifiers.

PROOF. According to LEMMA 2 in this case the two nodes periodically become simultaneous testers. If the faster node also has the greater identifier, each time simultaneous tests occur, that node becomes the tester. Assuming that simultaneous tests always occur, then the link is tested once every two testing intervals by the node with the greater identifier. Otherwise, if the faster node has the smaller identifier, each time simultaneous tests occur, this node turns out to be tested. In its next testing interval the faster node executes the test. Two testing intervals after that, simultaneous tests occur again, and this way successively. □

THEOREM 4. A *working* node running DNR tests an adjacent link once every two testing intervals, unless its clock is twice or more than twice slower than the clock of a *working* neighbor.

PROOF. If the clock of a *working* node is more than twice as fast as the clock of a *working* neighbor, and they are connected by a *working* communication link, the faster node is the only tester for that link, as stated by THEOREM 2. Thus, in this situation, the slower node never tests the link. If the clock of a *working* node is exactly twice as fast as the clock of a *working* neighbor, and they are connected by a *working* communication link, then either they alternate as tester and tested nodes after the occurrence of simultaneous tests or only the faster node tests the link, as stated by THEOREM 3.

On the other hand, if the clock speed of the node is less than twice slower than its neighbor's, then the node tests that adjacent link once every two testing intervals. This follows from the set of lemmas above. If a node recovers and finds out an adjacent link is *unresponsive*, LEMMA 1 proves that the node tests that link once every two testing intervals, as long as the link remains *unresponsive*. If a node recovers and finds out an adjacent link is *working*, LEMMA 3 proves the node tests that link once every two testing intervals, as long as the link remains *working*. If a node is *working* and an adjacent *working* link becomes *unresponsive*, the node continues testing that link once every two testing intervals as shown by LEMMA 4. At last, if a node is *working* and a healing event occurs at an *unresponsive* link, the node continues testing that link once every two testing intervals as shown by LEMMA 5.

Thus as these are the only possible cases, a *working* node running DNR tests an adjacent link once every two testing intervals, unless its clock is twice or more than twice slower than the clock of a *working* neighbor. □

For all proofs below we assume that the speeds of the clocks of any two neighbors are less than one the double of the other.

COROLLARY 1. If both nodes connected by a *working* link remain *working* for more than one testing interval, then only one test is executed on that link per testing interval as long as the nodes and the link remain *working*.

PROOF. This COROLLARY follows from THEOREM 4: as each *working* node running DNR tests an adjacent link once every two testing intervals, if the link remains *working*, the nodes must alternate their roles as tester and tested is successive intervals, so that only one test is executed on the link per testing interval. □

COROLLARY 2. If a node is the only *working* node adjacent to an *unresponsive* link for more than one testing interval, then only one test is executed on that link every two testing intervals as long as the link remains *unresponsive*.

PROOF. This COROLLARY also follows from THEOREM 4: as each *working* node running DNR tests an adjacent link once every two testing intervals, if a link remains *unresponsive* with only one adjacent *working* node, that node becomes the tester once each two testing intervals. □

Next we prove the worst-case event detection latency of the proposed testing strategy.

LEMMA 6. The *unresponsiveness* of a link is detected in at most two testing intervals.

PROOF. The *unresponsiveness* of a link is detected if at least one of its adjacent nodes is starting up, recovering or *working* while the link is or becomes *unresponsive*. If a node is starting up or recovering, the *unresponsiveness* of the link is detected as soon as it times out on the reply expected after it sends test requests to all of its neighbors.

If a node is *working* while a link becomes *unresponsive*, that node may be either the tester or the tested node in that testing interval. If it is the tester, it detects the *unresponsiveness* in at most one testing interval, as soon as it tests the link. Otherwise, if the *working* node is the tested node in that interval, that node will become the tester in the following interval, as stated by LEMMA 4, so it detects the *unresponsiveness* of the link in at most two testing intervals. □

It should be noted that above we prove the worst-case latency: information about a new event may be learned earlier, even from the dissemination algorithm.

LEMMA 7. A *healing* event is detected in at most two testing intervals.

PROOF. The healing of a link is detected by a *working* node if an adjacent node starts up or recovers while the link is recovering or *working*, or if a neighbor is *working*, while the *unresponsive* link to it recovers.

If a node starts up or recovers, it sends test requests to all of its neighbors. In this case, if the adjacent link is recovering or *working*, due to the *two-way* tests, as soon as the *working* neighbors reply, the state of the tester is detected as *working* as is *working* the state of the communication link to it.

If both adjacent nodes were *working* while the link recovered, those nodes were not alternating their roles as tester and tested nodes before the *healing* event. Thus, either the link may have a tester for that testing interval or not, because both nodes may be tested nodes at that interval. In this case the link will be tested only in the following testing interval, as stated by LEMMA 4. Because of the *two-way* testing strategy, the tested node also learns about the healing event. □

THEOREM 5. The event detection latency of DNR is two testing intervals in the worst-case.

PROOF. First consider a node running DNR which is either starting up, recovering or *working*. This node detects the *unresponsiveness* of a link either because an adjacent *working* node and/or link fail. The detection latency of these cases is at most two testing intervals, as stated by LEMMA 6.

The *healing* of a link is detected by a *working* node if an adjacent node starts up or recovers and the adjacent link is *working* or if a link recovers while the adjacent node is starting up, recovering or *working*. The detection latency of such events is also at most two testing intervals as stated by LEMMA 7.

As these are the only possible cases, the detection latency of DNR is at most two testing intervals. □

4 DISSEMINATION PHASE PROOFS

This section proves both the latency of information dissemination after the detection of an event and the correctness of the healing procedure. At first it is shown how the occurrence of events during a dissemination may affect its latency. Later we prove that the exchange of link state table information assures the correct dissemination of events in the case of a healing.

The latency of information dissemination in DNR is computed considering the time of the initiation of a dissemination after the detection of an event. Upon the detection of a new event on an adjacent link, a node running DNR starts the dissemination of new event information employing a parallel strategy. So, as a dissemination is started, a message is sent trough all adjacent links, and then it is forwarded by the *working* neighbors throughout the network in a similar way.

Thus a dissemination may follow multiple paths, so call a message *redundant* when it arrives at a node where it has already been disseminated. Let the *dissemination path* be the set of links over which the message is subsequently forwarded which reaches each node at first, i.e., before any redundant message. Call a *level* the set of nodes that are at the same number of hops distant from the starting node *in the dissemination path*.

DEFINITION 5. A *dissemination round* is defined as the time interval in which all nodes at one level of the dissemination path send their messages to all of the nodes in the next level.

Let the diameter of a connected component be the largest minimum distance between any two nodes in that component. The latency of information dissemination in a component is directly proportional to its diameter. It is important to notice that the diameter may change after the occurrence of new events.

We say that an event alters the diameter of a connected component *for a given dissemination* when it occurs in a link or node of that component not yet reached by the dissemination.

In the case of an *unresponsiveness* event, if it is not yet detected by the time the dissemination reaches it, it is detected by the dissemination itself. A healing event, on the other hand, is detected only by the arriving of a test request. Nevertheless, a healed link is used when the dissemination reaches it.

THEOREM 6. Consider that a dissemination is started upon the detection of an event and proceeds in a connected component where other events may occur. Consider the succession of those events that alter the diameter of the component for that dissemination. Let d be the diameter of the component after the last of such events occurs. It takes at most d dissemination rounds from the beginning of the dissemination until it reaches all nodes in the final component.

PROOF. As disseminations are propagated with a parallel strategy, in a connected component with diameter d the longest path has size at most d. So, according to the definition of a dissemination round above, it takes at most d dissemination rounds for the message being disseminated to reach all nodes in that component. □

We next prove the correctness of the healing procedure. We begin defining what a *healing message* is.

The message sent by the tested node to the tester upon the detection of a healing event is called a *healing message*. A healing message contains an extract of the *LinkStateTable* with all links whose timestamps are greater than 1.

Call a *healing procedure* the task of sending a healing message by the tested node to the tester and the subsequent sending of a dissemination message by the tested node for all its neighbors.

THEOREM 7. The *healing procedure* assures that complete information about the connected components previous to the healing arrive to the whole new component.

PROOF. The healing procedure is based on the assumption that the nodes that detected the healing have updated information about all events occurred in their connected components prior to the healing event. Indeed, if any of them does not have complete information about those events, that is because there are events still being disseminated. Those events will reach the healed link in the future and will be disseminated across it, thus arriving the whole new component, as long as no new events occur.

As the timestamps of unreachable links were reset in local link state tables of the previous components, the healing message generated contain only information about links located in the previous component of the sending node. According to the specification of the algorithm, the dissemination message sent by the node that receives the healing message is generated after updating the local *LinkStateTable* of that node and incrementing the timestamp of the healed link. Thus, that dissemination messsage contains information about both components that existed previously to the healing, plus information about the healing event itself.

So, eventhough part of the information contained in the final dissemination message is redundant in different portions of the network, it contains complete information about the new connected component generated with the healing. □

As a concluding remark, the dissemination latency of healing messages is equal to the dissemination latency proved in Theorem 6. In the case of the healing of a link that partitioned the network, the final diameter after the healing event is the sum of the diameters of each previous component. For a link that did not partition the network, its healing may decrease the final diameter (think of a ring), but the latency remains equal to the diameter. The same result applies to the case of a recovering node, once the healing of a node is equivalent to the healing of all its adjacent links.

5 Conclusion

In this paper we gave alternative specification and set of proofs for the Distributed Network Reachability algorithm which was originally presented in (DUARTE-JR; WEBER; FONSECA, 2012). The algorithm allows any node of a general topology network to compute which portions of the network are reachable and unreachable. Links are tested continually, at a testing interval, disseminating new information about events using a parallel strategy. The algorithm is capable of diagnosing dynamic events and allows network partitions and subsequent healings. At any time any working node may compute network reachability. We prove bounds on several properties of the algorithm.

Future work includes exploring several applications of this algorithm, both for computer networks as well as interconnection networks.

REFERENCES

BAGCHI, A.; HAKIMI, S. L. An optimal algorithm for distributed system-level diagnosis. In: TWENTY-FIRST INTL. FAULT TOLERANT COMPUTING SYMPOSIUM, 1991. **Proceedings of the...** New York, NY, USA, 2002. p. 214–221.

DUARTE-JR., E. P. et al. Non-broadcast network fault monitoring based on system-level diagnosis. In: IEEE/IFIP IM'97. **Proceedings of the...** San Diego, 1997. p. 597–609.

DUARTE-JR., E. P.; WEBER, A. A distributed network connectivity algorithm. In: IEEE/ISADS'03. **Proceedings of the...** New York, NY, USA, 2003. p. 285–292.

DUARTE-JR, E. P.; WEBER, A.; FONSECA, K. V. O. Distributed diagnosis of dynamic events in partitionable arbitrary topology network. **IEEE Trans. on Parallel and Distributed Systems**, v.23, p. 1415–1426, 2012.

MASSON, G.; BLOUGH, D.; SULLIVAN, G. **System Diagnosis**. In: FAULT-TOLERANT COMPUTER SYSTEM DESIGN. NJ, Prentice-Hal: Englewood Cliffs, 1996. p. 478–536.

RANGARAJAN, S.; DAHBURA, A. T.; ZIEGLER, E. A. A distributed system-level diagnosis algorithm for arbitrary network topologies. **IEEE Trans. Computers**, v. 44, p. 312–333, 1995.

SIQUEIRA, J. I.; FABRIS, E.; DUARTE-JR, E. P. A token based testing strategy for non-broadcast network diagnosis. In: FIRST IEEE LATIN AMERICAN TEST WORKSHOP. **Proceedings of the...** Rio de Janeiro, 2003. p. 166–171.

STAHL, M.; BUSKENS, R.; BIANCHINI, R. Simulation of the adapt on-line diagnosis algorithm for general topology networks. In: ELEVENTH IEEE SYMP. RELIABLE DISTRIBUTED SYSTEMS. **Proceedings of the...** [S.l.], 1992.

SUBBIAH, A.; BLOUGH, D. M. Distributed diagnosis in dynamic fault environments. **IEEE Transactions on Parallel and Distributed Systems**, v. 15, p. 453–467, n. 5, may. 2004.

WEBER, A.; DUARTE-JR., E. P.; FONSECA, K. V. O. An optimal test assignment for monitoring general topology networks. In: SEVENTH IEEE LATIN-AMERICAN TEST WORKSHOP. **Proceedings of the...** Buenos Aires, 2003. p. 131–136.