2009 29th IEEE International Conference on Distributed Computing Systems

A Distributed Termination Detection Algorithm for Dynamic Asynchronous Systems

Paul Johnson and Neeraj Mittal
Department of Computer Science
The University of Texas at Dallas
Richardson, TX 75080, USA
Emails: paul.johnson@student.utdallas.edu neerajm@utdallas.edu

Abstract

Termination detection in distributed systems has been a popular problem of study. It involves determining whether a computation running on multiple nodes has ceased *all its activities. A large number of termination detection algorithms have been proposed for static distributed systems in which the number of nodes present in the system is fixed and never changes during runtime. Recently, the termination detection problem has been investigated in the context of dynamic distributed systems in which individual nodes may join and/or leave the system at any time. In this paper, we propose an efficient algorithm for detecting termination of a computation in a dynamic, asynchronous, distributed system that allows nodes to join as well as leave the system while the computation is in progress. Our simulation results indicate that our algorithm has lower message complexity as well as lower detection latency than other comparable algorithms for solving the same problem.*

1. Introduction

In the termination detection problem, a set of nodes collectively execute a distributed computation. The computation is split into pieces. By assigning pieces to different nodes, the system throughput increases by allowing several tasks to be executed concurrently. A termination detection algorithm integrates with the system so that users can determine whether all nodes have finished their respective tasks without doing any book-keeping themselves.

An asynchronous message-passing distributed system consists of independent nodes. There is no notion of global time or global memory. Each node only knows its own local state and local time. Nodes communicate with each other via messages with arbitrary propagation delays. Further, nodes may have arbitrary execution speeds. These limitations make termination detection a non-trivial problem to solve.

Termination detection problem has been extensively studied for static distributed systems in which the set of nodes in the system is fixed in the beginning and does not change over time (e.g., [1], [2], [3], [4], [5], [6], see [7] for a survey). With the advent of new computing paradigms such as grid computing and peer-to-peer computing, *dynamic* distributed systems are becoming increasingly popular. In a dynamic distributed system, processes can join and leave the system at anytime. Consequently, the set of processes in the system may change with time. Dynamic distributed systems are especially useful for solving *large-scale problems* that require vast computational power. For example, distributed.net [8] has undertaken several projects that involve searching a large state-space to locate a solution. Some examples of such projects include RC5-72 to determine a 72-bit secret key for the RC5 algorithm, and OGR-25 to compute the Optimal Golomb Ruler with 25 and more marks.

Our contribution: In this paper, we propose a tree-based algorithm for termination detection in dynamic, asynchronous, distributed systems that allows nodes to join and leave the system while the computation is in progress. Our algorithm is derived from the tree-based algorithm for termination detection in static systems proposed by Dijkstra and Scholten in [1]. As in the case of Dijkstra and Scholten's algorithm, we assume that the computation is diffusing in nature, i.e., it originates from a single node. In contrast to our algorithm, in some (termination detection) algorithms for dynamic systems, nodes may not be able to leave the system until the computation has terminated [9], [10]. Further, in some algorithms, new nodes can only be spawned by existing nodes; nodes external to the system cannot join the system [11]. Our simulation results indicate that our algorithm has lower message complexity as well as lower detection latency than comparable termination detection algorithms [12].

Road map: The rest of this paper is organized as follows. In Section 2, we describe the previous work on termination detection in dynamic distributed systems. In Section 3, we describe both the system model assumed by our algorithm and the termination detection problem. We describe our termination detection algorithm in Section 4, prove its correctness in Section 5 and analyze its complexity in Section 6. Section 7 presents our simulation results. Finally, Section 8 concludes the paper.

2. Related Work

One of the earliest termination detection algorithms for dynamic distributed systems was proposed by Cohen and

1063-6927/09 $25.00 © 2009 IEEE
DOI 10.1109/ICDCS.2009.16

Lehmann in [9]. Similar to our algorithm, their algorithm is also derived from Dijkstra and Scholten's tree-based termination detection algorithm for static distributed systems [1]. Cohen and Lehmann's algorithm allows for nodes to join the system at any time, but makes no allowance for nodes to depart in a timely manner. Specifically, a node can leave the system only if it is a leaf node of the tree being maintained. As a result, in some cases, a departing node may be forced to wait until the termination is declared.

Dhamdhere et al. proposed a wave-based [2] termination detection algorithm in [12]. In the basic version of this algorithm, every computation message, also referred as *application message*, is acknowledged. An idle node initiates a termination detection wave once all its application messages have been acknowledged. The wave envelops a node only if it is idle and all its application messages have been acknowledged. Termination is detected when the wave returns to its initiator after visiting each node once. An optimized version was also proposed to eliminate the need for acknowledgments. In this version, the wave must propagate across every edge that carried an application message. Both of Dhamdhere et al.'s algorithms have high bit-message complexity, and high message processing time. Each termination detection message, also referred to as *control message*, carries a set of nodes. Therefore, in the worst case, bit message complexity of a control message is $O(N_{sys}B_{id})$ where N_{sys} denotes the number of nodes in the system, and B_{id} denotes the number of bits required to represent the identity of a node. Message processing time is also high because, for each control message reception, set operations such as set difference may be required, which can take $O(N_{sys}B_{id})$ time assuming that the sets are sorted.

Wang and Mayo proposed a termination detection algorithm in [10] that allows nodes to join the system at any time, but no node can leave the system until termination is detected. Further, the algorithm has very high (worst-case) detection latency. When a node that is already a member of the tree receives an application message, it joins the tree again, as a logically different node. This may cause the computation tree being maintained to become very long during normal execution since one node may appear multiple times in the tree, thereby leading to high detection latency.

Darling et al. proposed a termination detection algorithm in [13] that allows nodes to join and leave the system at any time. However, nodes are assumed to be equipped with "approximately synchronized" local clocks. Nodes are arranged in a logical ring and tokens are passed along the ring to detect termination. Synchronization of clocks imposes additional overhead on the system. Further, due to the arrangement of nodes in a logical ring, this algorithm has higher detection latency than tree-based algorithms.

DeMara et al. proposed a tiered termination detection algorithm in [11]. This algorithm relies on a fixed collection of *processing elements* (PEs) to convert the dynamic termination detection problem into a static one. The main idea is to keep track of the number of tasks produced and consumed on each PE. A controller maintains a table of each PE's values. PEs update the controller only upon becoming idle. The controller announces termination when the sum of all tasks produced equals the sum of all tasks consumed. This approach requires that a single PE (the controller) receive messages from all other PEs. As a result, the controller may become a bottleneck.

Peri and Mittal proposed an algorithm for detecting any stable property (including termination) in dynamic distributed systems in [14]. The algorithm maintains a spanning tree of all nodes in the system and uses the tree to repeatedly collect consistent snapshots of the system. The liveness of this algorithm is guaranteed only if nodes stop joining the system eventually. In a system in which nodes constantly join and leave, this algorithm may never terminate.

3. System Model and Problem Definition

3.1. System Model

We assume a dynamic asynchronous distributed system in which nodes communicate by exchanging messages with each other. There may be arbitrary message delays and execution speeds, but each must be finite. Nodes may join and leave the system at any time. A node may depart and rejoin, but upon rejoining, it acquires a new identity.

We assume that the system provides reliable, FIFO channels. Further, if a node p sends a message m to a departed node q, m is eventually delivered back to p with a special flag set to denote that m is a returned message.

3.2. Problem Definition

The distributed computation whose termination is to be detected is modeled as follows. At any given time, a node can be in either the *active* or *passive* state. Intuitively, an active node is involved in some computational work whereas a passive node is idle. An active node can send an application message to another node. A passive node becomes active on receiving an application message. An active node can become passive at any time. The computation is said to have terminated once all nodes have become passive and stay passive thereafter. For a static distributed system, this is equivalent to the condition that all nodes have become passive and all channels have become empty. Termination is known to be a *stable* property; once the computation terminates, it stays terminated.

In this paper, we focus on detecting termination of diffusing computations in which only a single node is active to begin with and that node is responsible for "spreading" the computation directly or indirectly to other nodes via application messages. We assume that the initiator of the

diffusing computation is *permanent*, i.e., the initiator belongs to the system in the beginning and never leaves the system. Note that an incoming node joins the system as a passive node. Also, we allow a node to depart from the system only if it is passive.

Two important complexity measures of a termination detection algorithm are *message complexity* and *detection latency*. Message complexity refers to the number of messages that the algorithm exchanges in order to detect the termination of the underlying computation. To account for message size, a related complexity measure of *bit-message complexity*, which counts the total number of bits used by the algorithm, is sometimes used. Detection latency refers to the time elapsed between when the computation terminates and when its termination is detected. To measure detection latency, we assume that message transmission time is $O(1)$ and message processing time is negligible. Clearly, it is desirable for a termination detection algorithm to have low message complexity as well as low detection latency.

4. Our Termination Detection Algorithm

The premise of our termination detection algorithm is as follows. The algorithm maintains a tree of all nodes currently participating in the computation rooted at the initiator of the computation. When an application message m is sent to a node p that is not a member of the tree, p attaches to the tree with the sender of m as its parent. Passive nodes detach from the tree in a way that does not violate termination detection safety. Termination is declared when the root node is the only member of the tree, and is passive.

4.1. Algorithm Terminology

A node first joins the system. As part of the system, it may receive application messages causing it to join the computation tree. After becoming part of the computation tree, the node may leave the tree at some later time. In fact, a node may join and leave the computation tree multiple times. Finally, the node may leave/depart from the system. To avoid the confusion, we use the phrases *attaching to the tree* and *detaching from the tree* to refer to joining and leaving the computation tree, respectively.

There are several variables that each node p stores:

1) **p.state** – state of the node, value can be *detached*, *active*, *passive*, or *detaching*. Its initial value is *detached*.
2) **p.detachFlag** – This flag denotes whether a node wishes to detach from the tree. It is set externally by the node itself (but not by our algorithm). Its initial value is $false$.
3) **p.detachMsgSent** – This denotes whether a node wishes to detach and has sent a detach message to its contact already. Its initial value is $false$.
4) **p.contact** – the contact of a node *p* is the first node *q* which sends an application message to *p*. This variable may change as p relocates within the tree.
5) **p.oldContact** – the previous contact of p. This variable is used during the relocate process to inform the old contact that this node has successfully relocated.
6) **p.friendSet** – when p sends an application message to another node q, q is added to *p.friendSet*.
7) **p.guestSet** – when p receives an ATTACH message from a node, p adds this node to its $guestSet$.
8) **p.grantedSet** – this set stores the guests that node *p* has given permission to detach. Node *p* cannot detach while this list is non-empty.
9) **p.timestamp** – denotes the number of application messages p has sent. It is incremented by 1 each time p sends an application message. Its initial value is 0.

The next two variables use the information contained in a message, which is explained shortly afterwards:

10) **p.receiveSet** – each entry in this set is stored as a tuple, $\langle node, timestamp \rangle$. When an application message is received, the tuple $\langle m.source, m.sentTS \rangle$ is added to *p.receiveSet*. If *p.receiveSet* already contains a tuple with *m.source* as the first element, the second element is replaced with $m.sentTS$.
11) **p.sentSet** – each entry in this set is stored as a tuple, $\langle node, timestamp \rangle$. When an application message m is sent, the tuple $\langle m.dest, p.timestamp \rangle$ is added to *p.sentSet*. If *p.sentSet* already contains a tuple with $m.dest$ as the first element, the second element replaced with the current value of $p.timestamp$.

Every message m sent by a node has the following fields:

1) **m.source** – the identifier of the node sending this message.
2) **m.dest** – the identifier of the node receiving of this message.
3) **m.recvTS** – the timestamp of the last message received from $m.dest$.
4) **m.sentTS** – the timestamp of the last message sent to $m.dest$.
5) **m.returnMessageFlag** – denotes whether this message is a returned message or not.

Every RELOCATE control message m (sent by a node detaching from the tree) has the additional field:

1) **m.newContact** – the node the receiver should attach to, if it wishes to remain a member of the computation tree

When we wish to specify the value associated with node p contained within a set S, we use the notation $S[p]$. Also, when an application message m is returned as undeliverable to node p, the message is delivered to the application. The application then decides what action to take for the returned message.

Since a node can choose to depart at any time, a node may want to depart, even if it is attached to the tree (but passive). Before leaving the system, it has to gracefully detach from the tree so that termination can still be detected in a safe manner. We want a node to be able to detach quickly so that the time a departing node has to stay in the system is minimized. We now describe (1) how a node may join and depart from the system, (2) when and how it may attach and detach from the tree, and (3) how the root node detects termination.

Procedure 1 On Sending Application Message m

```
1: update sentSet[m.source]
2: increment timestamp
3: if m.dest ∉ guestSet ⋃ friendSet then
4:     add m.dest to friendSet
```

Procedure 2 On Receiving Application Message m

```
1: update receiveSet[m.source]
2: if m is returned message then
3:     return m to application
4:     if sentSet[m.dest] = m.sendTS then
5:         /* This is the last application message we sent to m.dest. We
           will not get any more returned application messages. */
6:         remove m.dest from friendSet
7: else if state = detached then
8:     send(ATTACH) to m.source
9:     state := active
10:    contact := m.source
11:    deliver m
12: else if state = detaching then
13:    return m to sender
14: else
15:    deliver m
```

4.2. Joining the System

Nodes can join the system at any time. When a node joins the system, it registers with the name service. Once it registers with this service, other members of the system become aware of this node, and may start sending it application messages.

4.3. Attaching to the Tree

We use the following analogy to demonstrate the basic idea behind attach and detach operations.

Consider a dinner party in which the host wants to invite a large number of people, however he does not want to keep track of every guest. The host sends an invitation to a small number of friends and requests that they RSVP if they plan on attending. These friends, upon receiving their invitations think that it is a great idea, so they RSVP for the party and invite some of their own friends, also requesting an RSVP. Each person who accepts an invitation becomes a party guest. Each guest knows the person who first sent them an invitation, their *contact*, and knows all the *friends* they invited to the party, but not immediately which friends are their *guests*.

Let us now analyze how the attach operation works. A node attaches to the computation tree when it receives an application message and it is not already part of the tree.

If a node p sends an application message m to another node q, p executes Procedure 1. In this procedure, p first sets $p.sentSet[q] = p.timestamp$ and increments $p.timestamp$ by 1. Node p sets $m.sentTS = p.sentSet[q]$ and $m.recvTS = p.receiveSet[q]$ before finally sending message m. If q is not already a member of $p.friendSet$ or $p.guestSet$, p adds q to $p.friendSet$.

A node q, upon receiving an application message m, executes Procedure 2. The node first updates $q.receiveSet[p]$ and checks to see whether m is a returned (application) message. A returned message contains the exact same payload as the original message except it has a $returnMessage$ flag set to true. If m is a returned message, q returns m to the ap-

Procedure 3 On Receiving ATTACH Message m

```
1: if state ∈ {active, passive} then
2:     if friendSet contains m.source then
3:         remove m.source from friendSet
4:     add m.source to guestSet
5:     send(ACCEPT) to m.source
6:     if detachFlag AND NOT(detachMsgSent) then
7:         send(DETACH) to p.contact
8:         detachMsgSent := true;
9: else if state = detaching then
10:    if friendSet contains m.source then
11:        remove m.source from friendSet
12:        add m.source to guestSet
```

Procedure 4 On Receiving ACCEPT Message m

```
1: if Relocating then
2:     send(RELOCATED) to oldContact
3:     Relocating := false
```

Procedure 5 Function isLeafNode( )

```
1: return (guestSet = ∅) AND (friendSet = ∅)
```

Procedure 6 Function checkLeafNode( )

```
1: if guestSet = ∅ then
2:     for all q in friendSet do
3:         send(PROBE) to q
```

Procedure 7 On Becoming Passive

```
1: state := passive
2: checkLeafNode( )
3: if isLeafNode( ) then
4:     if amRoot then
5:         detect termination
6:     else
7:         send(DETACHED) to contact
8:         state := detached
```

Procedure 8 On Receiving DETACHED Message m

```
1: remove m.source from grantedSet
2: remove m.source from guestSet
3: if m.recvTS ≠ sentSet[m.source] then
4:     add m.source to friendSet
5: if state ∈ {detaching, passive} then
6:     checkLeafNode( )
7:     if isLeafNode( ) then
8:         send(DETACHED) to contact
9:         state := detached
10:    else if detachFlag then
11:        send(DETACH) to contact
12:        sentDetachMsg := true
```

Procedure 9 On Deciding to Detach

```
1: detachFlag := true
2: if (grantedSet ≠ ∅) OR Relocating then
3:     wait until (grantedSet = ∅) AND NOT(Relocating)
4:     send(DETACH) to contact
```

Procedure 10 On Receiving DETACH Message m

```
1: if (state = detaching) OR detachFlag then
2:     do nothing
3: else
4:     add m.source to grantedSet
5:     send(GRANT) to m.source
```

plication, then compares $m.sentTS$ to $q.sentSet[m.dest]$. If they are equal, then this is the last application message q sent to $m.dest$ and in this case, q acts as if m is a DECLINE message from $m.dest$ and executes Procedure 14. If m is not a returned message and q is already a tree member then it does not respond to m, but delivers m to the application. However, if q is not a tree member, it sets $q.contact = m.source$, $q.state = active$ and sends an ATTACH message to $m.source$.

A node p, upon receiving an ATTACH message from a node q, executes Procedure 3. It moves q from $p.friendSet$ to $p.guestSet$. If p is not currently in the *detaching* state it sends an ACCEPT message to q. If p is in the *detaching* state, it does not need to send an ACCEPT message because when it transitioned to the detaching state, it sent a RELOCATE message to q, which also acts as an implicit ACCEPT message. When node q receives the ACCEPT message it executes Procedure 4.

4.4. Detaching from the Tree

Continuing with our analogy, the host has a large house, enough to fit everyone who was invited comfortably. All guests enjoy the dinner party and time passes. Consider a particular guest, p, who has enjoyed the party, but has decided to return home. To make a decorous departure, this guest should first say farewell to his *contact*. Interrupting any conversation his *contact* is currently in would be rude, so the guest must first wait for the *contact* to become available. Once the *contact* is available, the guest is free to say his good-byes. During this farewell conversation, the topic of other guests comes up. The guest considers all of the *friends* that he invited. Any *friend* of this guest would

Procedure 11 On Receiving GRANT Message m

```
1: state := detaching
2: if isLeafNode( ) then
3:     send(DETACHED) to contact
4:     state := detached
5: else
6:     for all guest in guestSet ⋃ friendSet do
7:         send(RELOCATE, contact) to guest
```

Procedure 12 On Receiving RELOCATE Message m

```
1: if m is returned message then
2:     if sentSet[m.dest] = m.sendTS then
3:         remove m.source from friendSet
4:         checkLeafNode( )
5:         if isLeafNode( ) then
6:             send(DETACHED) to contact
7:             state := detached
8: else if m.source = contact then
9:     if (state = passive AND grantedSet = ∅) OR detachFlag
       then
10:        for all guest in guestSet ⋃ friendSet do
11:            send(RELOCATE, m.newContact) to guest
12:        state := detaching
13:    else
14:        send(ATTACH) to m.newContact
15:        Relocating := true
16:        oldContact := contact
17:        contact := m.newContact
18: else
19:    send(DECLINE) to m.source
```

Procedure 13 On Receiving RELOCATED Message m

```
1: remove m.source from guestSet
2: if m.recvTS ≠ sentSet[m.source] then
3:     add m.source to friendSet
4: checkLeafNode( )
5: if isLeafNode( ) then
6:     send(DETACHED) to contact
7: state := detached
```

feel uncomfortable if the guest left, leaving them alone in a house full of unfamiliar people, so the guest decides to introduce all his *friends* to his *contact*.

Some of his *friends* may have sent their RSVPs to another guest. In this case, the *friend* politely declines the introduction. However, *friends* who sent their RSVPs to this guest p, and thus are *guests* of p, will gladly accept the introduction. During the introduction the guest's *friend* and *contact* become acquainted. Once the guest p has introduced all of his *friends* to his *contact*, or they have declined the

Procedure 14 On Receiving DECLINE Message m

```
1: if m.sentTS = sentSet[m.source] then
2:     remove m.source from friendSet
3:     if isLeafNode( ) then
4:         if state ∈ {detaching, passive} then
5:             send(DETACHED) to contact
6:             state := detached
```

Procedure 15 On Receiving PROBE Message m

```
1: if m is returned message then
2:     if sentSet[m.dest] = m.sentTS then
3:         remove m.source from friendSet
4:         if isLeafNode( ) then
5:             if state ∈ {detaching, passive} then
6:                 send(DETACHED) to contact
7:                 state := detached
8: else if m.source ≠ p_i.contact then
9:     send(DECLINE) to m.source
```

Procedure 16 Termination Detection Condition

1: **if** $amRoot$ **AND** isLeafNode( ) **AND** $(state = passive)$ **then**
2: announce termination

introduction, the guest says a final farewell to his *contact* and is free to leave.

Let us now analyze how the detach operation works. Whenever a node p decides to detach, it will set $p.detachFlag = true$. There are three different ways a node can detach:

1) A node detects that it is a leaf node. (Procedure 7)
2) A node receives a GRANT message from its contact in response to its DETACH message. (Procedure 11)
3) A node receives a RELOCATE message from its contact who is itself detaching from the tree. (Procedure 12)

In the first case, a node p becomes passive with an empty $guestSet$. It first sends a PROBE message m to all nodes in its $friendSet$ by executing Procedure 6. Upon receiving a PROBE message m from node p, a node q executes Procedure 15. If m is a returned message, q treats it as if it were a DECLINE message sent from p by executing Procedure 14. Otherwise, if $q.contact = p$ then q is a $guest$ of p, so q ignores this message because there already is an ATTACH message in transit to p. If $q.contact \neq p$, q sends a DECLINE message, x, to p with $x.recvTS = q.receiveSet[p]$.

When node p receives a DECLINE message x from q, it first checks to see whether $x.recvTS = p.sentSet[q]$. If they are equal, it removes the sender from its $friendSet$. If they are not equal, there is at least one application message y, sent by p, that q had not received upon sending the DECLINE message. Furthermore, since channels are reliable and provide FIFO ordering, this DECLINE message could not have been sent in response to m. Since application message y may cause q to send an ATTACH message, p should ignore the stale DECLINE message x. Eventually, application message y and PROBE message m will arrive at node q, and it will generate a correctly timestamped reply. Once p receives responses (DECLINE messages or delayed ATTACH messages) for every PROBE message it sent (ignoring old messages) $p.friendSet$ becomes empty. If $p.guestSet$ is now non-empty, then p is not a leaf node, and does not detach unless the flag $p.detachFlag$ is true. If $p.detachFlag$ is true, the node departs using case 2. However, if the $p.guestSet$ is empty, then p is a leaf node and it sends a DETACHED message z to its contact where $z.recvTS = p.receiveSet[p.contact]$. Let r be $p.contact$, then upon receiving this message, r will execute Procedure 8 by checking if $z.recvTS = r.sentSet[p]$. If they are equal, r removes p from $r.guestSet$ and then the node is detached from the tree. If they are not equal, then there must be at least one application message that r has sent to p that p has not received yet. Node r will move p from $p.guestSet$ to $p.friendSet$ (p has detached, however it may re-attach when it receives the outstanding application message).

In order for the second case to occur, an interior node p becomes passive and decides that it wishes to detach. Node p sends a DETACH message to $p.contact$ and waits for a response. Let r be $p.contact$, then if r is departing, p eventually receives a RELOCATE message, and this becomes case 3. Otherwise, r immediately replies with a GRANT message and adds p to $r.grantedSet$. When p receives this message it executes Procedure 11 by sending a RELOCATE message m to all nodes in $p.guestSet$ and $p.friendSet$. Message m contains the variable $m.newContact = r$. When p receives a RELOCATED message it removes that node q from the appropriate set by executing Procedure 13. However, if p receives a DETACHED or DECLINE message d, it must compare $p.sentSet[q]$ to $d.timestamp$. If these two values are equal, then p removes this node from the appropriate set. If these values are not equal, then the message is old and is ignored. Once p receives a response from every node, $p.guestSet$ and $p.friendSet$ both become empty. At this point, this case is symmetric to case 1 when both sets are empty. Node p has become a leaf node, so it sends a DETACHED message to its contact.

The third case occurs when p's contact is itself detaching from the tree. If p receives a RELOCATE message m and is passive, it can decide to detach by executing Procedure 12. Node p then sends a RELOCATE message x to all nodes in its $guestSet$ and $friendSet$ where $x.newContact$ is set to $m.newContact$. Once this message is sent, this case is identical to the second case after p sends RELOCATE messages.

Example: To better understand the dynamics of the detach operation, consider the chain of nodes $p_0, p_1, \ldots, p_h$ where p_0 is the root node and p_h is a leaf node. Assume that the series of nodes $p_1, \ldots, p_h$ are passive and decide to detach from the tree. Node p_i with $1 \leqslant i \leqslant h$ sends a DETACH message to node p_{i-1}. In this case, node p_1 receives a GRANT message from node p_0 in response. This *single* GRANT message gives permission to p_1 and any of its descendants to detach. Any node who wishes to remain a member of the computation tree and is a descendant of p_1 may relocate to p_0. Node p_1 sends a RELOCATE message to its guests, including p_2. Node p_2 forwards this message to its $guestSet$ and $friendSet$. This chain of RELOCATE messages propagates downwards until it reaches p_h. Since p_h is a leaf node, it detaches by sending a DETACHED message to its contact. Node $p_{h-1}.guestSet$ and $p_{h-1}.friendSet$ eventually become empty and it sends its own DETACHED message. This chain of DETACHED messages continues back up to the root, allowing all nodes in the $p_1, \ldots, p_h$ chain to detach. Node p_1 detached using the Case 2 because it sent a DETACH message to p_0 and received a GRANT message in response. The rest of the nodes, $p_2, \ldots, p_h$ detached using Case 3 because they were

in the passive state and received a RELOCATE message.

Note that, depending on the exact scenario, it is possible for multiple nodes to detach from the tree concurrently. For instance, in the above example, multiple children of node p_1 can detach from the tree concurrently on receiving RELOCATE message from p_1.

4.5. Departing from the System

In order for a node to depart, it cannot be a member of the computation tree. Once this is true, a node contacts the name service and de-registers itself. At this time, the node has departed from the system.

4.6. Termination Detection

Continuing with our analogy, since the host's house is very large, it would be impractical for him to search the entire house to ensure that the party is indeed finished. Instead, the host only keeps track of the guests that he invited, or that were introduced to him. Once the host's list is empty, and he too wishes the party to finish, termination is detected.

The node p that initiated the diffusing computation declares termination by executing Procedure 16 whenever p becomes passive, or receives a control message that causes $p.friendSet$ or $p.guestSet$ to reduce in size. Only when $p.guestSet \bigcup p.friendSet = \emptyset$ and $p.state = passive$ will termination be announced.

5. Proof of Correctness

We show that the termination detection algorithm is safe (i.e., it announces termination only if the computation has indeed terminated) and live (i.e., once the computation terminates, it eventually announces termination). We also show that the detach operation is live (i.e., it eventually terminates). Due to space constraints, we only describe the main ideas here. Detailed proof of correctness can be found in [15].

5.1. No False Termination Detection

We say that a node p_k is *reachable* from the root node if there is a chain of processes $p_0, p_1, \ldots, p_k$, starting with the root node p_0 such that for each node p_j, where $0 \leqslant j < k$, $p_j.friendSet$ or $p_j.guestSet$ contains p_{j+1}. We next define an invariant as follows:

Invariant 1: If a node is active or has a non-empty incoming channel, then it is reachable from the root node.

We prove in [15] that our algorithm maintains the above invariant. Clearly, it follows from the invariant and Procedure 16 that the root node will not announce termination unless all nodes are passive and all channels are empty.

5.2. Eventual Termination Detection

To prove that our algorithm eventually detects termination, it is sufficient to show that once the computation has terminated, (1) the tree stops growing, and (2) the height of the tree reduces by one eventually. First, consider when nodes are added to the computation tree. A node that is part of the system, but not part of the computation tree attaches only when it receives an application message. Once termination has occurred, no application messages are sent. Therefore, at this time, the computation tree ceases to grow in size. To prove that the computation tree reduces its height by one, consider the checkLeafNode( ) function (Procedure 6). Upon becoming passive, a node checks to see if its $guestSet$ is empty. If the $guestSet$ is empty, then that node sends a PROBE message to each node in its $friendSet$. If this node is a leaf node, it receives DECLINE messages for every PROBE message sent. Once a node's $friendSet$ and $guestSet$ are empty and it is passive, it sends a DETACHED message to its contact. Once this message has been sent, the node has detached. Therefore, eventually all leaf nodes of the current computation tree detaches, reducing the height of the computation tree by one.

5.3. Liveness of Detach Operation

Whenever a node p decides to detach, it sends a DETACH message to $p.contact$. Node $p.contact$ might also be departing, and may wait before sending a response to p. Node $p.contact$'s contact may also be departing, and will wait before sending a response to $p.contact$. This waiting chain, however, has a finite length. Since the root node is permanent and never detaches from the tree, it will always grant any DETACH request it receives. Once this occurs, there is a chain of RELOCATE messages that propagates down until p receives this message. Node p will forward this RELOCATE message to its guests, and once every node in $p.guestSet$ and $p.friendSet$ responds, p becomes a leaf node and may depart. Therefore, the detach operation satisfies eventually terminates.

6. Complexity Analysis

The message complexity of our algorithm is dominated by DECLINE and RELOCATE messages. The number of DECLINE messages is bounded by $O(M_{app})$, where M_{app} denotes the number of application messages generated by the underlying computation. The number of RELOCATE messages is bounded by $O(\Delta N_{depart})$, where Δ denotes the maximum degree of any node and N_{depart} denotes the number of nodes that depart during the algorithm's execution. Therefore the message complexity of our termination detection algorithm is $O(M_{app} + \Delta N_{depart})$.

Detection latency can be bounded by $O(H_{max})$ where H_{max} denotes the maximum height of the computation tree. In Section 5.2, we show that once termination occurs, the tree height reduces by one and in [15] we show that this reduction in height only takes $O(1)$ time. As far as the time complexity of attach and detach operations is concerned, an attach operation has time complexity of $O(1)$ because each ATTACH message received immediately generates an ACCEPT message. However, detach operations have higher worst-case time complexity because there may be multiple nodes in a chain wishing to detach at the same time. However, this chain is bounded by $O(H_{max})$ because the root node never detaches from the tree and will always grant DETACH requests.

7. Simulation Results

Simulation experiments were performed using a custom discrete event simulator. To compare our algorithm's performance, we implemented the wave-based algorithm in [12]. In this paper, Dhamdhere et al. proposed a basic algorithm which requires every application message to be acknowledged. They also propose an optimized version which removes this requirement. We implemented the optimized version and refer to it as DIROpt. We refer to our algorithm as TreeDetect. We selected the DIROpt algorithm because it was one of the few algorithms that supports node departures during termination detection. Our focus for simulations was to compare algorithm performance in the presence of node joins/departures.

7.1. Experimental Methodology

To simulate an application upon which our algorithm detects termination, we created a class B application as described in [3]. This application maintains a variable workload which intuitively describes the amount of work needed by a node to complete some computation. As time progresses, each node decreases its workload and periodically sends application messages. Class B applications are those in which workload is distributed among all nodes in the system unequally and all nodes become passive at different times. We felt that this most accurately models an actual application. In a dynamic system, nodes are allowed to join and depart from the system at any time, making it unlikely that workload is equally distributed among all nodes.

One major difference between our simulations and the simulations in [3] is that, in our simulations, initially a single node has all the workload, and it is the responsibility of that node to distribute the workload among all other nodes. In [3], each node is provided with an initial workload, and these nodes randomly generate messages as workload is consumed. In our case, these application messages redistribute workload. If a node has above a certain amount of workload, it will redistribute it among other nodes with a certain probability. Once a node's workload reaches 0, it informs the termination detection algorithm that it has become passive.

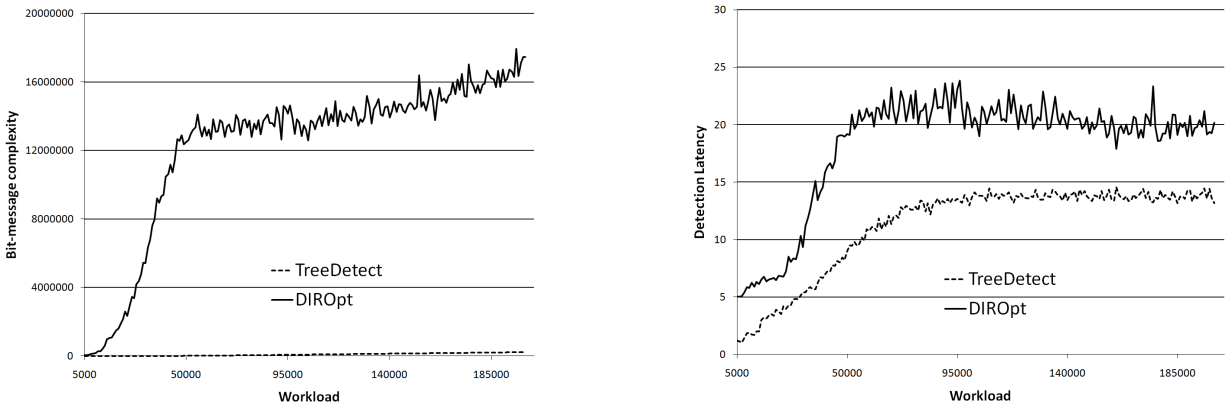
There are several defined parameters for each simulation run. Message propagation delay was modeled as an exponential distribution with a mean of 0.5. Nodes have a 0.06% chance to depart each time it completes a unit of work. This means that as the total initial workload increases in subsequent simulation runs, more nodes depart from the system. For example at 50,000 workload, an average of 30 nodes departed during the simulation; at 150,000 workload, 90 nodes departed. While completing a unit of work, each node also, independently, had a 5% chance to redistibute workload if its workload is larger than the workload threshold of 500. If a node decides to distribute workload, it splits the workload equally between 5 randomly selected neighbors and itself. To maintain an average number of nodes in the system, node arrivals were modeled as an exponential distribution with a mean inter-arrivial time based upon initial workload, initial number of nodes, and departure chance. The number of nodes in the system fluctuates, but on average stayed within 20% of the initial 100 nodes. All of these parameters remained constant with the exception of workload and node inter-arrival time. Results for a network of 100 nodes is shown. We also ran simulations with different values for several other parameters. Independently varrying the number of nodes, probability to distribute workload and departure probability showed similar results.

By varying the initial workload of the computation we were able to simulate increasingly computational-intensive applications. Simulation tests were performed as follows. For each workload value simulated, we performed 200 uniquely seeded simulations and took the average. We simulated workloads ranging from 5,000 to 200,000 in increments of 1,000.

7.2. Results

We compared bit-message complexity instead of message complexity of the two algorithms because it more accurately captures the overhead imposed by a termination detection algorithm. That being said, our algorithm still performed better than DIROpt in number of messages for all workload values, ranging from a 106% improvement to a 2050% improvement. In Figure 1(a), we can see the results for number of control bits used by the two termination detection algorithms. Our algorithm performs significantly better than DIROpt for all workload values. This is because, in DIROpt, termination detection messages contain lists of a node's neighbors. Our algorithm makes no such requirement, and only uses a small number of bits per control message.

Results for detection latency are shown in Figure 1(b). For all workload values, our algorithm detects termination



(a) Bit-message complexity

(b) Detection latency

Figure 1. Comparative evaluation of TreeDetect and DIROpt.

sooner than DIROpt.

8. Conclusion

In this paper, we have proposed a tree-based algorithm for detecting termination of diffusing computations in dynamic, asynchronous, distributed systems. In contrast to many existing termination detection algorithms, our algorithm allows nodes to join (even if external to the system) as well as leave the system while the computation is in progress. Our simulation results indicate that our algorithm has lower message complexity as well as lower detection latency than comparable termination detection algorithms. We plan to conduct more extensive simulations in the future. Preliminary tests with constant workload and varied departure probabilities in particular showed some interesting results.

References

[1] E. W. Dijkstra and C. S. Scholten, "Termination Detection for Diffusing Computations," *Information Processing Letters (IPL)*, vol. 11, no. 1, pp. 1–4, 1980.

[2] F. Mattern, "Algorithms for Distributed Termination Detection," *Distributed Computing (DC)*, vol. 2, no. 3, pp. 161–175, 1987.

[3] A. A. Khokhar, S. E. Hambrusch, and E. Kocalar, "Termination Detection in Data-Driven Parallel Computations/Applications," *Journal of Parallel and Distributed Computing (JPDC)*, vol. 63, no. 3, pp. 312–326, Mar. 2003.

[4] N. R. Mahapatra and S. Dutt, "An Efficient Delay-Optimal Distributed Termination Detection Algorithm," *Journal of Parallel and Distributed Computing (JPDC)*, vol. 67, no. 10, pp. 1047–1066, Oct. 2007.

[5] N. Mittal, S. Venkatesan, and S. Peri, "A family of optimal termination detection algorithms," *Distributed Computing (DC)*, vol. 20, pp. 141–162, Jun. 2007.

[6] M. J. Livesey, R. Morrison, and D. S. Munro, "The Doomsday Distributed Termination Detection Protocol," *Distributed Computing (DC)*, vol. 19, no. 5–6, pp. 419–431, Apr. 2007.

[7] J. Matocha and T. Camp, "A Taxonomy of Distributed Termination Detection Algorithms," *Journal of Systems and Software (JSS)*, vol. 43, no. 3, pp. 207–221, Nov. 1998.

[8] distributed.net, http://www.distributed.net/.

[9] S. Cohen and D. Lehmann, "Dynamic Systems and their Distributed Termination," in *Proceedings of the 1st ACM Symposium on Principles of Distributed Computing (PODC)*, Ottawa, 1982, pp. 29–33.

[10] X. Wang and J. Mayo, "A General Model for Detecting Termination in Dynamic Systems," in *Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS)*, Santa Fe, New Mexico, Apr. 2004.

[11] R. F. DeMara, Y. Tseng, and A. Ejnioui, "Tiered Algorithm for Distributed Process Quiescence and Termination Detection," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 18, no. 11, pp. 1529–1538, Nov. 2007.

[12] D. M. Dhamdhere, S. R. Iyer, and E. K. K. Reddy, "Distributed Termination Detection for Dynamic Systems," *Parallel Computing*, vol. 22, pp. 2025–2045, 1997.

[13] D. Darling, J. Mayo, and X. Wang, "Stable Predicate Detection in Dynamic Systems," in *Proceedings of the International Conference on Principles of Distributed Systems (OPODIS)*, 2005.

[14] S. Peri and N. Mittal, "Monitoring Stable Properties in Dynamic Peer-to-Peer Distributed Systems," in *Proceedings of the 25th IARCS Annual Conference on the Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, R. Ramanujam and S. Sen, Eds., Hyderabad, India, Dec. 2005, pp. 420–431.

[15] P. Johnson and N. Mittal, "A Distributed Termination Detection Algorithm for Dynamic Asynchronous Systems," Department of Computer Science, The University of Texas at Dallas, Tech. Rep. UTDCS-06-09, Mar. 2009.