ELSEVIER

The Journal of Systems and Software 43 (1998) 207–221

The Journal of Systems and Software

A taxonomy of distributed termination detection algorithms

Jeff Matocha [1], Tracy Camp *

Department of Computer Science, The University of Alabama, P.O. Box 870290, Tuscaloosa, AL 35487-0290, USA

Received 1 December 1996; received in revised form 10 April 1997; accepted 10 May 1997

Abstract

An important problem in the field of distributed systems is that of detecting the termination of a distributed computation. Distributed termination detection (DTD) is a difficult problem due to the fact that there is no simple way of gaining knowledge of the global state of the system. Of the algorithms proposed in the last 15 years, there are many similarities. We have categorized these algorithms based on the following factors: algorithm type (e.g., wave, credit-recovery), required network topology, algorithm symmetry, required process knowledge, communication protocol (synchronous or asynchronous), communication channel behavior (first-in first-out (FIFO) or non-FIFO), message optimality, and fault tolerance. This methodology is intended to guide future research in DTD algorithms (since research continues on these algorithms) as well as to provide a classification survey for this area.

1. Introduction

In the field of distributed systems, the problem of detecting the termination of a distributed computation has been well studied. Distributed termination detection (DTD) was introduced in 1980 independently by Francez (Francez, 1980) and Dijkstra and Scholten (Dijkstra and Scholten, 1980). Since its introduction, and especially during the mid-1980s, DTD has been a popular problem to research. Although it is not popular as a research topic today, one or two new algorithms a year continue to be presented. DTD is difficult to achieve efficiently as it is a global state detection problem. The global state of a distributed system consists of states of each of the processes in the system and the states of the channels in the system. Because processes in a distributed system do not share clocks or memory, synchronization problems make the detection of global state difficult. An algorithm which accomplishes global state determination has been presented by Chandy and Lamport (Chandy and Lamport, 1985), and termination detection has been solved using this global state approach (Misra and Chandy, 1982; Misra, 1983; Chandy and Misra, 1985, 1986a). Of course, there are many DTD algorithms that are not based on the global state approach by Chandy and Lamport.

DTD has also been viewed as a problem of system consensus. All processes in the system must be ready to terminate and all processes must then agree (either by some information gathering or by information passing) that the computation is terminated. In this paper, however, we shall informally show that DTD is a harder problem than consensus (see Section 2.8).

Researchers have placed the problem of DTD in the larger category of quiescence detection (Chandy and Misra, 1986a; Mattern, 1989). Quiescence is defined as the state in which there are no messages in the system and all processes are waiting. Both termination and deadlock fit under the heading of quiescence and are treated similarly in some of the proposed algorithms. The problems are similar since they both concern sensing when the system is in a state where there is no way to continue processing.

A distributed system is defined as a set of n processes, $P = \{p_0, p_1, \ldots, p_n\}$, which are distributed across a network, and a set of communication channels, E, by which the processes can transfer information. The communication channels are sometimes considered as directed channels in DTD algorithms, but are most often bidirectional. This set theoretic definition of a distributed system can be naturally represented as a graph. Thus, processes are often discussed as nodes or vertices and communication channels are often discussed as edges.

* Corresponding author. Tel.: +1 205 348 9516; fax: +1 205 348 0219; e-mail: cap@cs.ua.edu.

[1] E-mail: camp@cs.ua.edu

0164-1212/98/$ – see front matter
PII: S0164-1212(98)10034-1

We make the following assumptions about any distributed system.

- There is no shared memory. Hence, information that is to be shared must be transmitted from one process to another by a message along some channel in E.
- There is no common clock. For example, there is no way to ensure that all processes perform some action at exactly three o'clock.
- Communication takes arbitrary, but finite, time. In other words, a process cannot predict when a message is to be delivered and, therefore, the sender does not know when the receiving process is cognizant of the information within the message.

The computation that is being executed by the distributed system is called the *basic computation*. Messages that are involved in this basic computation are called *basic messages*. We define M to be the number of basic messages used in the computation. Processes in the system can be in either of two states: *active*, in which the process is currently working on the basic computation, or *passive*, in which the process is currently waiting to be either activated or terminated. We define a model of a distributed system to constrain the action of active and passive processes. Processes behave according to the following rules.

1. Initially, each process in the system is either active or passive.
2. An active process may become passive at any time.
3. Only an active process may send a basic message to another process.
4. A passive process can only become active if it receives a basic message.

The above assumptions ensure that the system behaves in a predictable manner. If one can determine that all processes in the system are passive and there are no basic messages in transit, one has detected termination. These two conditions are the necessary and sufficient conditions for detecting termination given the four rules of distributed systems behavior above.

In order to discern the conditions for DTD, each process executes an algorithm in addition to its basic computation. The DTD algorithm runs concurrently, but does not interfere, with the basic computation. Running this DTD algorithm introduces more messages into the system. The messages used expressly for detecting termination are called *control messages*. Control messages are sent and received by both active and passive processes, but do not modify their active or passive status.

The algorithms which have been presented in the literature have been proposed in a logical progression throughout the years. In this paper, a categorization of the past work in DTD algorithms is presented. We classify the algorithms based on similarities, and we illustrate how each algorithm fits each category. The categories that we define to classify the DTD algorithms are intended to aid in developing future algorithms, for furthering the study of this type of algorithm, as well as for choosing a DTD algorithm for use in one's own implementation.

We define the following categories for classifying DTD algorithms in this paper.

1. Type of algorithm – For example, a wave algorithm is a popular type of DTD algorithm.
2. Necessary network topology – For those algorithms where it is necessary to exploit the underlying topology of the network in order to detect termination correctly.
3. Algorithm symmetry – If an algorithm is symmetric, each process runs an identical algorithm.
4. Process knowledge – For example, a process that requires information about the network in order to perform its duty, has special knowledge.
5. Communication protocol – Each algorithm assumes either synchronous or asynchronous communication.
6. Message arrival – Each algorithm assumes either first-in first-out (FIFO) message channels or no restrictions on the arrival of messages.
7. Message optimality – If an algorithm, in its worst case, uses the number of messages which researchers have proven to be a lower bound on the number of messages necessary to detect termination, it is considered message optimal.
8. Fault tolerance – If a system can detect termination when there are portions of the system that do not work as expected, it is considered fault tolerant.

We define the eight categories and classify the current DTD algorithms into these categories in Section 2. We discuss each of the categories in detail as well as the motivation for choosing them. Within the discussion, we indicate any dependencies of a classification; for example, the fact that synchronous communication implies FIFO message ordering. In Section 3, we conclude the paper with a discussion of the research possibilities in the area.

2. Classification categories

In studying DTD algorithms, we noted eight areas which are useful in comparing the proposed algorithms in the literature. In the following sections, each of the eight categories is described with a discussion of how a proposed algorithm relates to the category. The type of algorithm category is helpful for classifying any type of algorithm, distributed or otherwise. Communication protocol, message arrival, and constraints on the network are similar in that they describe constraints which the algorithm places on its execution environment. Message optimality is a measure of algorithmic complexity, which is always an important item to inspect. Process symmetry and process knowledge are similar

and are considered by some to be a single classification unit. Lastly, any real-world distributed system must be concerned with fault tolerance. These categories, though chosen in this paper for DTD algorithms, are general enough to apply to any distributed algorithm.

2.1. Type of algorithm

In order to classify a DTD algorithm by type, one must determine the general action of the algorithm. DTD algorithms range in type from theoretically based methods to ad hoc methods. In a few cases, a method is derived from other distributed problems (e.g., global state detection); typically, however, a method has been developed specifically for the problem of DTD.

The most popular method of constructing DTD algorithms has been to create a wave algorithm. A wave algorithm is one in which a message is passed to each node in the graph (directly or indirectly) by a single initiator or a set of initiators. Each process then returns information to the initiator(s). These "waves" of message passing collect information about the global state of the system. The nature of wave algorithms makes them applicable to detecting termination.

Tel (1994) formally defines a wave algorithm as one that satisfies the following three requirements:

1. The wave must terminate.
2. The initiator must make a decision.
3. The decision is preceded by an event in each process.

These conditions guarantee that there is a decision after *something* happens in each process. The only way to ensure each process does something is to pass a wave throughout the network. The information which is collected during the wave is a piece of the global state, although it may be inconsistent. Since DTD is a global state detection problem, it is natural that wave algorithms are popular for the proposed solutions.

Typically, wave algorithms exploit a feature of the network's topology. First, we discuss waves in a network in which there is a cycle that contains each process in P. These cyclic algorithms are simple examples of the wave algorithm type. One early example of this type is presented in Dijkstra et al. (1983). For the token and each node in the system, there is a color attribute, which is set initially to white. When the initiator (p_0) becomes passive, it sends a white token to its successor (p_{n-1}). When the current token holder (p_i) becomes passive, it passes the token to its successor (p_{i-1}). If p_i is a white node, it passes the token with whatever color it received. If p_i is a black node, it passes a black token. A node becomes black if it sends a basic message to another node. In addition to being initially white, each process becomes white upon forwarding the token to its successor. When p_0 receives a white token, is white itself, and is passive, termination has been detected. Note that in this algorithm:

1. each wave eventually terminates;
2. p_0 makes a decision after collecting information (i.e., the token color); and
3. p_i $(0 \leqslant i < n)$ receive and pass the token prior to the decision.

There are other wave algorithms that take advantage of a logical cycle within the network (Francez et al., 1981; Rana, 1983; Arora and Sharma, 1983; Kumar, 1985; Mattern, 1987a; Muller, 1987; Sanders, 1987; Hazari and Zedan, 1987; Mayo and Kearns, 1994). The network topology is further restricted for the algorithm in (Misra, 1983). In this case, the network is assumed to be a token ring network; thus, no other communication channels between processes are available. DTD is much simpler to determine in this restricted graph than in an arbitrary graph.

A tree network is another logical topology that is naturally exploited for wave algorithms. That is, a tree must be transposed upon the existing network. A wave algorithm in a logical tree network is typically one in which the root initiates the computation. The algorithm splits the token into pieces, one piece for each child, during the wave down the tree, and then combines the pieces of the token into a single token during the wave up the tree. When a non-leaf node collects tokens from each of its children, it sends a single token to its parent node. At the root of the tree, a piece of the global state is known when the root has collected tokens from all its children. Due to the transitive nature of the algorithm, the root can determine when all other nodes have passed their tokens. In Francez's seminal paper (Francez, 1980), a tree-based wave algorithm is described. Upon the initiation of the check for termination, the tree is "frozen" by a wave, thus pausing the basic computation. The root, upon receiving knowledge that all nodes are frozen, passes another wave to detect a *consistent* global state of the system. Naturally, the freezing aspect is detrimental to the basic computation. Therefore, more recent tree based wave algorithms do not use freezing to obtain a consistent global state of the system (Francez and Rodeh, 1982; Topor, 1984; Lai, 1986).

A clique, also called a complete graph, is a graph which has connections from each node to every other node. On these graphs, one can superimpose a cycle or a tree on which to run a wave algorithm. The algorithm presented in (Szymanski et al., 1985a, b) requires a clique graph as a topology and uses a tree-based wave algorithm. Of course, requiring a clique is a larger restriction than requiring that a tree is superimposed upon the network. There are two algorithms that use wave algorithms without making any assumptions about the underlying topology of the network (Chandy and Misra, 1985; Mattern, 1987b). Of course, algorithms that make no assumption about the network are the most flexible wave algorithms.

Wave algorithms are elegant and natural solutions to DTD, but there is one property of wave algorithms that

make them unattractive for implementation. Typically wave algorithms are repetitive, thus making them inefficient. That is, the DTD algorithm sends wave after wave until termination is detected. Many of the above algorithms have a message complexity of $O(M \times n)$ in the worst case. (Recall that M is the number of basic messages and n is the number of processes in the distributed system.) In other words, there is possibly one wave for *every* message in the basic computation.

Two wave algorithms do not exhibit the repetitive nature of the previously mentioned algorithms. In these non-repetitive wave algorithms, the initiator is different for each wave (Huang, 1988; Haldar and Subramanian, 1988). When a process becomes passive, it initiates a wave to detect the termination status of the system by sending a token around a cycle or a tree. Upon receipt of a token at an active process, the token is purged. When the last active process becomes passive, it senses, by receiving tokens from all processes either directly or indirectly, that the entire system is passive. The average case message complexity for these non-repetitive forms of wave algorithms is less than the repetitive wave algorithms. Unfortunately, the worst case continues to be $O(M \times n)$.

Although the majority of distributed termination detection algorithms are wave algorithms, other types of DTD algorithms exist. One such type of non-wave algorithms is what we call parental responsibility algorithms. This type of DTD algorithm builds trees based on the message passing of the basic computation (*computation trees*). The definition of passive is extended in these algorithms; a parent cannot become passive until all of its children are passive. When the root of the computation tree becomes passive, it can determine that termination has occurred.

Dijkstra and Scholten's parental responsibility algorithm (Dijkstra and Scholten, 1980) infers information about other processes by requiring that when process p_i activates process p_j $(i \neq j)$, p_i becomes responsible for p_j. When p_j becomes passive, it reports its new status to p_i. Obviously, p_i cannot become passive until all of the processes for which it is responsible have become passive. With Dijkstra and Scholten's parental responsibility algorithm, the root is the last process waiting to become passive; when the root becomes passive, termination is detected. This algorithm, though one of the seminal DTD algorithms, is message optimal. That is, the message complexity is $O(M)$. (See Section 2.7 for further discussion of optimal message complexity.) Other algorithms that use a parental responsibility algorithm are (Misra and Chandy, 1982; Cohen and Lehmann, 1983; Lai and Wu, 1995).

One algorithm uses a combined technique of parental responsibility and wave algorithms (Shavit and Francez, 1986). The algorithm is defined for a forest. Each tree in the forest runs the Dijkstra and Scholten parental responsibility algorithm; a wave is circulated among the roots of trees in the forest in order to detect termination of all the trees.

Another clever way to detect termination, which does not fall into the above two types, is the credit/recovery or weight throwing algorithm. These algorithms stem from a similar technique in distributed garbage collection. Mattern presents an algorithm that begins with one total credit in the distributed system (Mattern, 1989). Each active process holds a portion of the credit. When process p_i communicates with process p_j $(i \neq j)$, p_i sends p_j a fraction, f, of the credit it holds $(0 < f < p_i$'s holding). When a process becomes passive, it sends the credit it holds to an active process. When a process holds exactly one credit and becomes passive, it detects that termination has occurred.

This elegant credit/recovery solution has been used in several proposed algorithms (Rokusawa et al., 1988; Venkatesan, 1989; Tseng, 1995). These algorithms are subject to problems in their implementation due to the divisions of the credit. In (Huang, 1989), there is a description of a "space-efficient scheme to encode the weights".

Table 1 lists all the algorithms that we classify in this paper along with their associated type. The column labeled "Other" in Table 1 is for algorithms which we have not discussed in this section, as the type of these algorithms is ad hoc. We explore other aspects of these four algorithms in later sections.

2.2. Network topology

In many distributed termination detection algorithms, it is sometimes assumed that a particular topology is present in the network. If a certain topology is present, it is easier to write a correct and efficient DTD algorithm. Many DTD algorithms make assumptions about the topology of the communication network for this reason.

The most common structure which algorithms expect is a Hamiltonian cycle. A graph contains a Hamiltonian cycle if there is a cycle which uses no edge more than once and contains each process in P exactly once. Many algorithms use a Hamiltonian cycle superimposed upon the existing network as the DTD network on which the control messages travel (Francez et al., 1981; Dijkstra et al., 1983; Arora and Sharma, 1983; Kumar, 1985; Mattern, 1987a; Muller, 1987; Sanders, 1987; Hazari and Zedan, 1987; Mayo and Kearns, 1994). A Hamiltonian cycle is a topological trait that lends itself to wave algorithms, a popular DTD algorithm type as illustrated in Section 2.1. Finding a Hamiltonian cycle in a graph has been proven to be NP-completem (Cormen et al., 1990), thus creating problems for the use of these algorithms. Other algorithms require that the network is configured as a ring, which is a trivial Hamiltonian cycle

Table 1
Algorithms and their associated type

Algorithm	Cyclic wave	Tree wave	General wave	Non-repetitive wave	Parental responsibility	Credit/ recovery	Other
(Francez, 1980)		X					
(Dijkstra and Scholten, 1980)					X		
(Francez et al., 1981)	X						
(Misra and Chandy, 1982)					X		
(Francez and Rodeh, 1982)		X					
(Cohen and Lehmann, 1983)					X		
(Dijkstra et al., 1983)	X						
(Rana, 1983)	X						
(Misra, 1983)	X						
(Arora and Sharma, 1983)	X						
(Topor, 1984)		X					
(Szymanski et al., 1985a)		X					
(Szymanski et al., 1985b)		X					
(Kumar, 1985)	X						
(Chandy and Misra, 1985)			X				
(Lai, 1986)		X					
(Shavit and Francez, 1986)		X			X		
(Skyum and Eriksen, 1986)							X
(Mattern, 1987a)	X						
(Mattern, 1987b)			X				
(Muller, 1987)	X						
(Sanders, 1987)	X						
(Hazari and Zedan, 1987)	X						
(Rokusawa et al., 1988)						X	
(Huang, 1988)				X			
(Haldar and Subramanian, 1988)				X			
(Mattern, 1989)						X	
(Venkatesan, 1989)						X	
(Huang, 1989)						X	
(Chandrasekaran and Venkatesan, 1990)							X
(Ye and Keane, 1991)							X
(Lai et al., 1992)							X
(Mayo and Kearns, 1994)	X						
(Tseng, 1995)						X	
(Lai and Wu, 1995)					X		

(Rana, 1983; Misra, 1983; Haldar and Subramanian, 1988).

Some algorithms use a tree that is built from the computation (a *computation tree*), where the calling process is the parent and the called process is the child (Francez, 1980; Dijkstra and Scholten, 1980; Misra and Chandy, 1982; Francez and Rodeh, 1982; Cohen and Lehmann, 1983; Mattern, 1987a, 1989; Lai and Wu, 1995). Computation tree algorithms use the same channels for control communication that are used for basic messages, except in the opposite direction. Therefore it is assumed that the channels are bidirectional. Computations which operate as defined here are called *diffusing* because they begin with a single process (the root) and then branch out from it. All basic computations may not lend themselves to diffusing algorithms. Therefore, algorithms that require diffusion place restrictions on the kinds of computations with which they may execute.

As mentioned in Section 2.1, the algorithm from Shavit and Francez uses a combined approach (Shavit and Francez, 1986). This algorithm combines the parental responsibility algorithm of Dijkstra and Scholten (Dijkstra and Scholten, 1980) with a cycle-based wave algorithm. Each tree in the forest runs Dijkstra and Scholten's algorithm and a wave gathers information from the roots to detect termination. The combined approach allows the computation to be non-diffusing. In diffusing computations, Shavit and Francez's algorithm will exhibit the optimal message complexity as exhibited by Dijkstra and Scholten's algorithm. In a forest of diffusing computations, the complexity of the wave algorithm is inherent. Therefore, the generality gained by the combined approach (over Dijkstra and Scholten's approach) increases the message complexity.

Many algorithms have no requirement on the network topology. Included are algorithms which require a spanning tree. Early algorithms discuss spanning trees that are created at compile-time. Spanning tree creation can be done easily at run-time, however, and therefore require no preset tree in the network. Algorithms needing a spanning tree include (Topor, 1984; Lai, 1986;

Venkatesan, 1989; Chandrasekaran and Venkatesan, 1990; Lai et al., 1992). Algorithms that have no requirement on the network topology (and do not create a spanning tree) include (Szymanski et al., 1985a, b; Chandy and Misra, 1985; Skyum and Eriksen, 1986; Mattern, 1987b; Rokusawa et al., 1988; Huang, 1988, 1989; Ye and Keane, 1991; Tseng, 1995). Table 2 lists all the DTD algorithms discussed in this paper and the associated network topology requirement.

2.3. Algorithm symmetry

A DTD algorithm is symmetric if each process executes an identical algorithm. Symmetric algorithms are those in which no process is distinguished from the others for any purpose. Thus, if a DTD algorithm uses a unique identification for the processes, the algorithm is considered asymmetric. One can argue, however, that this type of asymmetric algorithm can become symmetric by checking a condition to distinguish the process identification. Another example of asymmetry in a DTD algorithm is one with a set of leaf processes, a set of interior processes, and a single root process, as in (Francez, 1980; Francez and Rodeh, 1982; Topor, 1984; Lai, 1986); in this case, a test of node type must be made and appropriate code executed. We consider any information to distinguish a process as *special* process knowledge (see Section 2.4). We designate these algorithms as "Specialized" in Table 3.

The largest percentage of asymmetric DTD algorithms are asymmetric due to one process being designated (before run-time) as the initiator/detector (Dijkstra and Scholten, 1980; Misra and Chandy, 1982; Cohen and Lehmann, 1983; Dijkstra et al., 1983; Rokusawa et al., 1988; Mattern, 1989; Tseng, 1995; Lai and Wu, 1995). Often, the initiator/detector is merely the root of the tree, which is natural for algorithms that use computation trees. Huang proposes an algorithm in which a central agent is chosen before run-time (Huang, 1989). This algorithm, as well as the initiator/detector algorithms, requires asymmetry throughout its execution.

Algorithms that require a spanning tree are asymmetric due to a distinguished root node (Venkatesan,

Table 2
Algorithms and their necessary network topology

Algorithm	Hamiltonian cycle	Computation tree	Spanning tree	No requirement	Other
(Francez, 1980)		X			
(Dijkstra and Scholten, 1980)		X			
(Francez et al., 1981)	X				
(Misra and Chandy, 1982)		X			
(Francez and Rodeh, 1982)		X			
(Cohen and Lehmann, 1983)		X			
(Dijkstra et al., 1983)	X				
(Rana, 1983)	X				
(Misra, 1983)	X				
(Arora and Sharma, 1983)	X				
(Topor, 1984)			X		
(Szymanski et al., 1985a)					X
(Szymanski et al., 1985b)					X
(Kumar, 1985)	X				
(Chandy and Misra, 1985)					X
(Lai, 1986)			X		
(Shavit and Francez, 1986)	X				
(Skyum and Eriksen, 1986)					X
(Mattern, 1987a)	X	X			
(Mattern, 1987b)				X	
(Muller, 1987)	X				
(Sanders, 1987)	X				
(Hazari and Zedan, 1987)	X				
(Rokusawa et al., 1988)				X	
(Huang, 1988)				X	
(Haldar and Subramanian, 1988)	X				
(Mattern, 1989)		X			
(Venkatesan, 1989)			X		
(Huang, 1989)				X	
(Chandrasekaran and Venkatesan, 1990)			X		
(Ye and Keane, 1991)				X	
(Lai et al., 1992)			X		
(Mayo and Kearns, 1994)	X				
(Tseng, 1995)					X
(Lai and Wu, 1995)		X			

Table 3
Algorithms and their process symmetry

Algorithm	Specialized	p_0 only	p_0 at run time	Token	Symmetric
(Francez, 1980)	X				
(Dijkstra and Scholten, 1980)		X			
(Francez et al., 1981)					X
(Misra and Chandy, 1982)		X			
(Francez and Rodeh, 1982)	X				
(Cohen and Lehmann, 1983)		X			
(Dijkstra et al., 1983)		X			
(Rana, 1983)					X
(Misra, 1983)				X	
(Arora and Sharma, 1983)				X	
(Topor, 1984)	X				
(Szymanski et al., 1985a)					X
(Szymanski et al., 1985b)					X
(Kumar, 1985)				X	
(Chandy and Misra, 1985)				X	
(Lai, 1986)	X				
(Shavit and Francez, 1986)				X	
(Skyum and Eriksen, 1986)					X
(Mattern, 1987a)					X
(Mattern, 1987b)					X
(Muller, 1987)					X
(Sanders, 1987)				X	
(Hazari and Zedan, 1987)					X
(Rokusawa et al., 1988)		X			
(Huang, 1988)					X
(Haldar and Subramanian, 1988)					X
(Mattern, 1989)		X			
(Venkatesan, 1989)			X		
(Huang, 1989)		(central agent)			
(Chandrasekaran and Venkatesan, 1990)			X		
(Ye and Keane, 1991)				X	
(Lai et al., 1992)			X		
(Mayo and Kearns, 1994)					X
(Tseng, 1995)		X			
(Lai and Wu, 1995)		X			

1989; Chandrasekaran and Venkatesan, 1990; Lai et al., 1992). Asymmetry in this case is similar to the algorithms with an initiator/detector or a central agent. In all three of these asymmetric algorithms, bottleneck problems exist. Also, fault tolerance is difficult since the distinguished node is important and would be difficult to recover after failure.

Other algorithms are initially asymmetric, to ensure that there is a single token in the network (Misra, 1983; Arora and Sharma, 1983; Kumar, 1985; Chandy and Misra, 1985; Shavit and Francez, 1986; Sanders, 1987; Ye and Keane, 1991). These algorithms start the computation with an initiator holding the token. Following the initialization, there is no asymmetry.

Algorithms which are symmetric are more flexible than asymmetric algorithms. Symmetric algorithms include (Francez et al., 1981; Rana, 1983; Szymanski et al., 1985a, b; Skyum and Eriksen, 1986; Mattern, 1987a, b; Muller, 1987; Hazari and Zedan, 1987; Huang, 1988; Haldar and Subramanian, 1988; Mayo and Kearns, 1994). Symmetry is important to allow less reliance on a special process in the system. Moving from asymmetry to symmetry also brings an algorithm a step closer to a fault tolerant solution (see Section 2.8). Table 3 lists each DTD algorithm with its associated symmetry characteristic.

2.4. Process knowledge

Many DTD algorithms assume that the processes have knowledge of the system initially. The information typically required is necessary at compile-time and causes the algorithm to be less general. For example, an algorithm may require knowledge of the static size of the network. Requiring this type of information initially is detrimental to the algorithm's usefulness, since it is compile-time knowledge and restricts the network of processes from changing.

In a ring network, it is a typical assumption that a process knows its successors (Francez et al., 1981; Dijkstra et al., 1983; Rana, 1983; Misra, 1983; Arora and Sharma, 1983; Kumar, 1985; Mattern, 1987a, b;

Muller, 1987; Sanders, 1987; Hazari and Zedan, 1987; Haldar and Subramanian, 1988; Mayo and Kearns, 1994). In (Arora and Sharma, 1983), the authors also require that each process has knowledge of the (shortest) distance to every other process in the system.

Algorithms using tree structures often require all processes to know whether they are a root, leaf, or interior node as well as the process's parent and children (Francez, 1980; Dijkstra and Scholten, 1980; Francez and Rodeh, 1982; Topor, 1984; Lai, 1986; Venkatesan, 1989; Chandrasekaran and Venkatesan, 1990; Lai et al., 1992). In algorithms that use a computation tree, the node type information is determined, and will change, during the execution of the computation. In addition to this dynamic information requirement, (Misra and Chandy, 1982) requires knowledge of a process's neighbors. Inferring neighbors from the network should be simple.

Due to the central agent in Huang's algorithm (Huang, 1989), it is different from previously mentioned algorithms. Huang requires that all processes in the system know which process is the central agent. Requiring this information is not the same as in algorithms which have an initiator/detector; algorithms do not require that a process (other than the initiator/detector itself) knows which process is the initiator/detector.

In a few cases, algorithms require processes to know global information about the network environment. Three algorithms require that all processes know an upper bound on the diameter of the network (Szymanski et al., 1985a, b; Skyum and Eriksen, 1986). Huang requires that each process has a list of all the processes in the system (Huang, 1988).

Lamport's logical clocks (Lamport, 1978) have greatly affected distributed systems, including DTD algorithms. Two algorithms require logical clocks (Rana, 1983; Mayo and Kearns, 1994). As opposed to Rana's clock (Rana, 1983), Mayo and Kearns create an algorithm for "roughly synchronized clocks". In this algorithm, δ represents the maximum time that any two clocks may be apart in the entire system.

There are many algorithms that require no information about the system environment (Cohen and Lehmann, 1983; Shavit and Francez, 1986; Rokusawa et al., 1988; Mattern, 1989; Ye and Keane, 1991; Tseng, 1995; Lai and Wu, 1995). Of course, algorithms which need no system information are the most general algorithms in terms of special knowledge. Table 4 lists all algorithms and their required process knowledge.

2.5. Communication protocol

Early DTD algorithms were based on Hoare's Communicating Sequential Processes (CSP) notation (Hoare, 1978). Since CSP is a synchronous protocol for message passing, these early algorithms are elegant DTD algorithms (Francez, 1980; Francez et al., 1981; Misra and Chandy, 1982; Francez and Rodeh, 1982; Cohen and Lehmann, 1983). Synchronous message passing is equivalent to requiring message travel to be instantaneous. Thus, CSP-based DTD algorithms need not be concerned with the possibility that there are messages in transit, thereby reducing the necessary and sufficient conditions for DTD to merely deciding if all processes in the system are passive. Elegant solutions are possible with synchronous communication compared to asynchronous communication. The instantaneous communication granted by CSP further ensures that all messages arrive in the order in which they are sent (well-ordered message passing). This ordering is a stronger property than the FIFO message ordering that we discuss in the next section. Other algorithms require synchronous message passing, but are not written in CSP (Dijkstra et al., 1983; Arora and Sharma, 1983; Topor, 1984; Szymanski et al., 1985a, b; Mayo and Kearns, 1994).

Realizing that synchronous communication is restrictive, some authors write their algorithm in CSP, yet prove their algorithm correct in the asynchronous case. Rana's algorithm is written in CSP, although a section is included which states how the algorithm is "easily modified for" asynchronous message passing (Rana, 1983). Chandy and Misra present an algorithm which is expressed in CSP, but can execute in asynchronous systems (Chandy and Misra, 1985).

Although synchronous communication can be modeled in an asynchronous system, it is not considered a better technique. The overhead of creating a synchronous system is great and, therefore, an asynchronous system is preferred. Algorithms with asynchronous communication are more flexible, realistic, and efficient than those that use synchronous communication. Asynchronous communication algorithms include (Dijkstra and Scholten, 1980; Misra, 1983; Kumar, 1985; Lai, 1986; Shavit and Francez, 1986; Skyum and Eriksen, 1986; Mattern, 1987a, b; Muller, 1987; Sanders, 1987; Hazari and Zedan, 1987; Rokusawa et al., 1988; Huang, 1988; Haldar and Subramanian, 1988; Mattern, 1989; Venkatesan, 1989; Huang, 1989; Chandrasekaran and Venkatesan, 1990; Ye and Keane, 1991; Lai et al., 1992; Tseng, 1995; Lai and Wu, 1995). Asynchronous communication has the attribute that messages have arbitrary (but finite) time of travel in the system. Detecting whether there are any messages in transit is a difficult task in a DTD algorithm. In Table 5, we list the DTD algorithms and the protocol they need for communication.

2.6. Message arrival

If a channel is required to deliver messages in the order in which they are sent, it is called a FIFO channel. If a FIFO channel is required by a DTD algorithm, the solution to detecting termination is easier to create. A

Table 4
Algorithms and their process knowledge

Algorithm	Successors	Node information	Upper bound on net diameter	Other	No information
(Francez, 1980)		X			
(Dijkstra and Scholten, 1980)		X			
(Francez et al., 1981)	X				
(Misra and Chandy, 1982)		X			
(Francez and Rodeh, 1982)		X			
(Cohen and Lehmann, 1983)					X
(Dijkstra et al., 1983)	X				
(Rana, 1983)	X			logical clocks	
(Misra, 1983)	X				
(Arora and Sharma, 1983)	X			distance function	
(Topor, 1984)		X			
(Szymanski et al., 1985a)			X		
(Szymanski et al., 1985b)			X		
(Kumar, 1985)	X				
(Chandy and Misra, 1985)	X				
(Lai, 1986)		X			
(Shavit and Francez, 1986)					X
(Skyum and Eriksen, 1986)			X		
(Mattern, 1987a)	X				
(Mattern, 1987b)	X				
(Muller, 1987)	X				
(Sanders, 1987)	X				
(Hazari and Zedan, 1987)	X				
(Rokusawa et al., 1988)					X
(Huang, 1988)				list of p_is	
(Haldar and Subramanian, 1988)	X				
(Mattern, 1989)					X
(Venkatesan, 1989)		X			
(Huang, 1989)				central agent	
(Chandrasekaran and Venkatesan, 1990)		X			
(Ye and Keane, 1991)					X
(Lai et al., 1992)		X			
(Mayo and Kearns, 1994)	X			logical clocks	
(Tseng, 1995)					X
(Lai and Wu, 1995)					X

few asynchronous DTD algorithms require FIFO channels (Misra, 1983; Skyum and Eriksen, 1986; Hazari and Zedan, 1987; Haldar and Subramanian, 1988; Venkatesan, 1989; Chandrasekaran and Venkatesan, 1990). A FIFO channel is also implicit in any synchronous communicating systems; see Section 2.5 for a list of the synchronous DTD algorithms. A FIFO channel can be achieved with the network protocol (e.g. TCP). This protocol ensures that messages, although not always received in order, are delivered to the application in a FIFO order.

A protocol written for non-FIFO channels is more general since it will execute on either a non-FIFO or FIFO channel with no modifications. Unrestricted algorithms include (Dijkstra and Scholten, 1980; Kumar, 1985; Lai, 1986; Shavit and Francez, 1986; Mattern, 1987a, b; Muller, 1987; Sanders, 1987; Rokusawa et al., 1988; Huang, 1988; Mattern, 1989; Huang, 1989; Ye and Keane, 1991; Lai et al., 1992; Tseng, 1995; Lai and Wu, 1995). The algorithm in (Chandy and Misra, 1985) is unique. The algorithm does not require FIFO channels, however, it uses a message to "flush the channel" in order to ensure that no messages are in transit. In Table 6, we list the DTD algorithms and classify them according to their message arrival properties.

2.7. Message optimality

Chandy and Misra proved in 1986 that there is a worst case lower bound on the number of control messages used by any DTD algorithm with asynchronous communication (Chandy and Misra, 1986b). They proved that the best asynchronous solutions to the DTD problem have a worst case (lower bound) of $\Omega(M)$ control messages, where M is the number of basic messages in the computation. This bound means that for each message sent in the basic computation, there is a constant number of control messages to determine when termination has occurred. Interestingly, one of the first proposed algorithms used exactly M control messages (Dijkstra and Scholten, 1980). Other early algorithms used extremely high numbers of control messages. Wave

Table 5
Algorithms and their communication protocol

Algorithm	Synchronous Communication	Asynchronous Communication
(Francez, 1980)	X (CSP)	
(Dijkstra and Scholten, 1980)		X
(Francez et al., 1981)	X (CSP)	
(Misra and Chandy, 1982)	X (CSP)	
(Francez and Rodeh, 1982)	X (CSP)	
(Cohen and Lehmann, 1983)	X (CSP)	
(Dijkstra et al., 1983)	X	
(Rana, 1983)	X (CSP)	"easily modified for"
(Misra, 1983)		X
(Arora and Sharma, 1983)	X	
(Topor, 1984)	X	
(Szymanski et al., 1985a)	X	
(Szymanski et al., 1985b)	X	
(Kumar, 1985)		X
(Chandy and Misra, 1985)		X (though in CSP)
(Lai, 1986)		X
(Shavit and Francez, 1986)		X
(Skyum and Eriksen, 1986)		X
(Mattern, 1987a)		X
(Mattern, 1987b)		X
(Muller, 1987)		X
(Sanders, 1987)		X
(Hazari and Zedan, 1987)		X
(Rokusawa et al., 1988)		X
(Huang, 1988)		X
(Haldar and Subramanian, 1988)		X
(Mattern, 1989)		X
(Venkatesan, 1989)		X
(Huang, 1989)		X
(Chandrasekaran and Venkatesan, 1990)		X
(Ye and Keane, 1991)		X
(Lai et al., 1992)		X
(Mayo and Kearns, 1994)	X	
(Tseng, 1995)		X
(Lai and Wu, 1995)		X

algorithms, with their repetitive nature, use many more messages than the minimum (typically $O(M \times n)$). Each wave is $O(n)$ since each process must answer to the initiator. For each basic message, of which there are M, a single wave might be used. Other algorithms that exhibit message optimality in terms of M are (Cohen and Lehmann, 1983; Mattern, 1989; Venkatesan, 1989; Huang, 1989; Chandrasekaran and Venkatesan, 1990; Lai et al., 1992). Two algorithms by Mattern (Mattern, 1987a, b) exhibit $O(M)$ behavior in graphs that are stars or are complete. Lai and Wu present a fault tolerant algorithm which exhibits optimal behavior in the case that there are no faults (Lai and Wu, 1995). All other algorithms do not exhibit optimal behavior.

Chandrasekaran and Venkatesan proved in 1990 that there is another lower bound for the number of control messages to detect termination (Chandrasekaran and Venkatesan, 1990). They proved that any algorithm must use "at least $|E|$ messages, where $|E|$ is the number of links in the network". They combine the results of their proof with the results from (Chandy and Misra, 1986b) to state that "any asynchronous algorithm that solves the distributed termination detection problem uses at least $\Omega(|E| + M)$ messages in the worst case". Therefore, all algorithms listed above that fulfill $O(M)$ control messages, fulfill this bound if $M > |E|$ (a natural assumption).

DTD algorithms use the number of messages required as a measure of complexity. One might consider message length to be a better measure of complexity; however, the difference in transmission time between a message of $O(1)$ bits and $O(n)$ bits is negligible. The time taken to access the transmission medium, on the other hand, is much greater than the actual transfer time. Table 7 lists the algorithms according to their message optimality. As can be seen in the table, most algorithms are not optimal in the number of control message transmissions.

2.8. Fault tolerance

Fault tolerance is the requirement that an algorithm continues to work regardless of problems in the network or at individual nodes. This robust action of an

Table 6
Algorithms and their restrictions on message arrival

Algorithm	FIFO	No restriction
(Francez, 1980)	X (Synchronous)	
(Dijkstra and Scholten, 1980)		X
(Francez et al., 1981)	X (Synchronous)	
(Misra and Chandy, 1982)	X (Synchronous)	
(Francez and Rodeh, 1982)	X (Synchronous)	
(Cohen and Lehmann, 1983)	X (Synchronous)	
(Dijkstra et al., 1983)	X (Synchronous)	
(Rana, 1983)	X (Synchronous)	
(Misra, 1983)	X	
(Arora and Sharma, 1983)	X (Synchronous)	
(Topor, 1984)	X (Synchronous)	
(Szymanski et al., 1985a)	X (Synchronous)	
(Szymanski et al., 1985b)	X (Synchronous)	
(Kumar, 1985)		X
(Chandy and Misra, 1985)	X (Used to flush channels)	
(Lai, 1986)		X
(Shavit and Francez, 1986)		X
(Skyum and Eriksen, 1986)	X	
(Mattern, 1987a)		X
(Mattern, 1987b)		X
(Muller, 1987)		X
(Sanders, 1987)		X
(Hazari and Zedan, 1987)	X	
(Rokusawa et al., 1988)		X
(Huang, 1988)		X
(Haldar and Subramanian, 1988)	X	
(Mattern, 1989)		X
(Venkatesan, 1989)	X	
(Huang, 1989)		X
(Chandrasekaran and Venkatesan, 1990)	X	
(Ye and Keane, 1991)		X
(Lai et al., 1992)		X
(Mayo and Kearns, 1994)	X (Synchronous)	
(Tseng, 1995)		X
(Lai and Wu, 1995)		X

algorithm is especially important when considering distributed systems which may be executing on a wide area network. With the explosive growth of the Internet, widely distributed computations continue to be an important research topic. Clearly, DTD with asynchronous communication is a difficult problem in the midst of faulty system components. In 1985, Fischer et al. proved that it is not always possible to gain consensus of the reliable processes in a system, even if the processes are trying to agree on something as simple as a binary number (Fischer et al., 1985). It has also been proven in (Koo and Toueg, 1988) that it is not possible to gain global common knowledge of the simple binary number in a system with the weaker "transient faults", i.e., faults that occur for an arbitrary, but finite length of time. Therefore, the standard model of a distributed system is often modified in order to make detection of termination in a faulty system possible.

The fail-stop fault model is one with the assumption that any node that fails will immediately stop upon failure and not transmit or receive any messages from that point forward. An extended statement of the necessary and sufficient conditions for termination is made for the cases with fail-stop faults in the system. Termination is now defined as the state in which all processes are either passive *or faulty* and there are no *deliverable* messages in the system. There have been four fault tolerant algorithms based on the fail-stop model (Misra, 1983; Venkatesan, 1989; Tseng, 1995; Lai and Wu, 1995). An extra requirement necessary to gain global common knowledge is that faulty processes are assumed to be detectable within a finite period of time. Furthermore, messages that are *undeliverable* (directed toward a faulty process) are either discarded or are returned to the sender.

The fail-stop fault tolerant algorithm proposed by Lai and Wu (Lai and Wu, 1995) is a modification of Dijkstra and Scholten's parental responsibility algorithm (Dijkstra and Scholten, 1980) described in Section 2.1. The authors of the fault tolerant algorithm begin by noting the problems that must be solved in order to produce a fault tolerant algorithm:

1. Root failures must be followed by some non-faulty process being designated as the new coordinator.

Table 7
Algorithms and their message optimality

Algorithm	Optimal	Not optimal
(Francez, 1980)		X
(Dijkstra and Scholten, 1980)	X	
(Francez et al., 1981)		X
(Misra and Chandy, 1982)		X
(Francez and Rodeh, 1982)		X
(Cohen and Lehmann, 1983)	X	
(Dijkstra et al., 1983)		X
(Rana, 1983)		X
(Misra, 1983)		X
(Arora and Sharma, 1983)		X
(Topor, 1984)		X
(Szymanski et al., 1985a)		X
(Szymanski et al., 1985b)		X
(Kumar, 1985)		X
(Chandy and Misra, 1985)		X
(Lai, 1986)		x
(Shavit and Francez, 1986)		X
(Skyum and Eriksen, 1986)		X
(Mattern, 1987a)	X (If a star or complete graph)	X (Otherwise)
(Mattern, 1987b)	X (If a star or complete graph)	X (Otherwise)
(Muller, 1987)		X
(Sanders, 1987)		X
(Hazari and Zedan, 1987)		X
(Rokusawa et al., 1988)		X
(Huang, 1988)		X
(Haldar and Subramanian, 1988)		X
(Mattern, 1989)	X	
(Venkatesan, 1989)	X (If a constant number of failures)	X (Otherwise)
(Huang, 1989)	X	
(Chandrasekaran and Venkatesan, 1990)	X	
(Ye and Keane, 1991)		X
(Lai et al., 1992)	X	
(Mayo and Kearns, 1994)		X
(Tseng, 1995)		X
(Lai and Wu, 1995)	X (If no failures)	X (If failures)

2. A non-faulty process needs to know if it has received all messages from a process that has since become faulty.
3. Processes that become faulty may be responsible for a tree of processes. This tree of processes must be reassigned to a non-faulty parent.

The first problem is solved by ensuring the non-faulty process with the lowest index is designated as the coordinator. The second problem is solved by a FAIL-FLUSH service the authors assume is present in the network protocol. This service is initiated by a non-faulty process which flushes the messages from the channel for a specified process (known to have failed). The third problem is the most complicated. The authors specify that a child of a failed process, upon realizing its parent is faulty, adopts the coordinator as its new parent. A simple adoption technique is not enough to ensure fault tolerance, since message delays can cause the coordinator to detect false termination before the child's message arrives. Further details on solving this problem are covered (Lai and Wu, 1995).

Byzantine failures are failures which cause processes or channels to lie to the others in the network. This type of failure model can stem from a process that transmits corrupted information or a channel that is exposed to an electromagnetic force which causes it to transmit incorrect information. This model of failure makes more complex problems than the fail-stop model of failure. There have been no DTD algorithms that deal with Byzantine failures in any capacity. An algorithm for DTD with Byzantine failures would have to use duplicate information in order to determine which (if any) processes are lying. Lamport et al. prove that the problem of getting processes to agree on a single value in a distributed system with Byzantine failures is only possible if more than two-thirds of the processes are "loyal" (Lamport et al., 1982). With unforgeable messages using unique signatures, it is possible to get all loyal processes to agree. The problem in termination detection is that not only must the processes agree, they must agree that they agree. The algorithm needs a way of detecting the lying processes and ignoring what these

processes send regarding the consensus. In looking at the complexity of DTD, the DTD problem is understood to be more difficult than consensus. Research on Byzantine fault tolerant DTD algorithms is an open problem, even in the case where serious constraints to the system are placed on the Byzantine model of failure. Table 8 lists the DTD algorithms and if they are fault tolerant. Misra's fault tolerant algorithm (Misra, 1983) does not handle fail-stop or Byzantine failures with the processes, but this wave algorithm does handle the loss of the token.

3. Conclusions and future research

We have defined eight categorization criteria for distributed termination detection algorithms in this paper. The type of algorithm is an overview of how the algorithm works. Network topology, communication protocol, and message arrival all concern themselves with how the system works. Algorithm symmetry and process knowledge are items that consider the generality of the algorithm. The complexity of the algorithm is measured by the message optimality criteria. Fault tolerance is a classification that addresses the implementation of the algorithm. These categories, though chosen in this paper for DTD algorithms, are general enough to apply to any distributed algorithm.

We also give a categorization of the work to date on DTD algorithms. This work should guide future research on the problem of distributed termination detection. This exposition characterizes the current distributed termination detection algorithms and illustrates where further work is needed. From the tables in this paper, we can summarize the preferred characteristics of a DTD algorithm in each category and then suggest future research goals. Although there is no best algorithm type, a wave algorithm usually carries along with it a repetitive nature and, therefore, a non-optimal message complexity. Of course, an algorithm that makes no assumptions about the network topology is most general, as are totally symmetric algorithms. An algorithm that requires no process knowledge is preferred for run-time dynamics. Asynchronous communication is by far the protocol

Table 8
Algorithms and their fault tolerance

Algorithm	Fault tolerant	Not fault tolerant
(Francez, 1980)		X
(Dijkstra and Scholten, 1980)		X
(Francez et al., 1981)		X
(Misra and Chandy, 1982)		X
(Francez and Rodeh, 1982)		X
(Cohen and Lehmann, 1983)		X
(Dijkstra et al., 1983)		X
(Rana, 1983)		X
(Misra, 1983)	X (Token regeneration)	
(Arora and Sharma, 1983)		X
(Topor, 1984)		X
(Szymanski et al., 1985a)		X
(Szymanski et al., 1985b)		X
(Kumar, 1985)		X
(Chandy and Misra, 1985)		X
(Lai, 1986)		X
(Shavit and Francez, 1986)		X
(Skyum and Eriksen, 1986)		X
(Mattern, 1987a)		X
(Mattern, 1987b)		X
(Muller, 1987)		X
(Sanders, 1987)		X
(Hazari and Zedan, 1987)		X
(Rokusawa et al., 1988)		X
(Huang, 1988)		X
(Haldar and Subramanian, 1988)		X
(Mattern, 1989)		X
(Venkatesan, 1989)	X	
(Huang, 1989)		X
(Chandrasekaran and Venkatesan, 1990)		X
(Ye and Keane, 1991)		X
(Lai et al., 1992)		X
(Mayo and Kearns, 1994)		X
(Tseng, 1995)	X	
(Lai and Wu, 1995)	X	

of choice. An algorithm which places no restriction on the arrival of messages is most realistic for the behavior of channels. A message optimal algorithm is preferred, as is the robustness of a fault tolerant algorithm. Therefore, a non-wave, symmetric, asynchronous, non-FIFO, message optimal, fault tolerant algorithm with no assumption about the network topology and no necessary process knowledge, if one exists, would be a huge development in this field. Furthermore, Byzantine failures have not been considered in any proposed algorithms and is an open topic of research. Any solution would be welcomed regardless of restrictions.

In looking at the papers which have been presented in this paper, we select a few that deserve special note. The algorithm by Venkatesan (Venkatesan, 1989) is a credit/recovery algorithm which uses a spanning tree (and therefore knows its parent and children as well as p_0 at run-time), requires asynchronous communication, is fault tolerant, and is message optimal if there is a constant number of faults. Our only criticism of this DTD algorithm is that it requires FIFO message channels. Requiring FIFO channels, however, is a minor complaint compared to the benefits of this algorithm. Lai and Wu's algorithm (Lai and Wu, 1995) makes no restriction on message arrival, uses a computation tree for its parental responsibility algorithm, needs no process knowledge, uses asynchronous message passing, and is fault tolerant. Our only criticism with this algorithm is that it is message optimal only if there are no faults in the system. Again, however, this is a minor complaint. An algorithm that is message optimal and fault tolerant probably does not exist. Of course, these two algorithms are not as elegant as some of the initial algorithms (due to their attempts to be fault tolerant). For an elegant solution, we note Dijkstra and Scholten's seminal algorithm. Dijkstra and Scholten's algorithm (Dijkstra and Scholten, 1980) is a parental responsibility algorithm (requiring a p_0 before run-time for its computation tree), uses asynchronous communication, has no message arrival restrictions, and is optimal. Although this algorithm is not fault tolerant, it is one of the first algorithms in the literature. Many of the algorithms proposed after this one pale in comparison.

References

Arora, R.K., Sharma, N.K., 1983. A methodology to solve the distributed termination problem. Inform. Systems 8 (1), 37–39.

Chandrasekaran, S., Venkatesan, S., 1990. A message-optimal algorithm for distributed termination detection. Journal of Parallel and Distributed Computing 8, 245–252.

Chandy, K.M., Lamport, L., 1985. Distributed snapshots: Determining global states of distributed systems. ACM Trans. Comput. Systems 3 (1), 63–75.

Chandy, K.M., Misra, J., 1985. A paradigm for detecting quiescent properties in distributed computations. In: Apt, K.R. (Ed.), Logics and Models of Concurrent Systems. Springer, New York.

Chandy, K.M., Misra, J., 1986a. An example of stepwise refinement of distributed programs: Quiescence detection. ACM Trans. Programming Languages and Systems 8 (3), 326–343.

Chandy, K.M., Misra, J., 1986b. How processes learn. Distributed Computing 1 (1), 40–52.

Cohen, S., Lehmann, D., 1983. Dynamic systems and their distributed termination. In: ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing, pp. 29–33.

Cormen, T.H., Leiserson, C.E., Rivest, R.L., 1990. Introduction to Algorithms. McGraw-Hill, New York.

Dijkstra, E.W., Feijen, W.H.J., van Gasteren, A.J.M., 1983. Derivation of a termination detection algorithm for distributed computations. Inform. Process. Lett. 16 (5), 217–219.

Dijkstra, E.W., Scholten, C.S., 1980. Termination detection for diffusing computations. Inform. Process. Lett. 11 (1), 1–4.

Fischer, M.J., Lynch, N.A., Paterson, M.S., 1985. Impossibility of distributed consensus with one faulty process. J. ACM 32 (2), 374.

Francez, N., 1980. Distributed termination. ACM Trans. Programming Languages and Systems 2 (1), 42–55.

Francez, N., Rodeh, M., 1982. Achieving distributed termination without freezing. IEEE Trans. Software Eng. 8 (3), 287–292.

Francez, N., Rodeh, M., Sintzoff, M., 1981. Distributed termination with interval assertions. In: Formalization of Programming Concepts, Lecture Notes in Computer Science, vol. 107. Springer, New York.

Haldar, S., Subramanian, D.K., 1988. Ring based termination detection algorithm for distributed computations. Inform. Process. Lett. 29 (3), 149–153.

Hazari, C., Zedan, H., 1987. A distributed algorithm for distributed termination. Inform. Process. Lett. 25 (5), 293–297.

Hoare, C.A.R., 1978. Communicating sequential processes. Commun. ACM 21 (8), 666–677.

Huang, S., 1988. A fully distributed termination detection scheme. Inform. Process. Lett. 29 (1), 13–18.

Huang, S., 1989. Detecting termination of distributed computations by external agents. In: IEEE Nineth International Conference on Distributed Computer Systems, pp. 79–84.

Koo, R., Toueg, S., 1988. Effects of message loss on the termination of distributed protocols. Inform. Process. Lett. 27 (4), 181–188.

Kumar, D., 1985. A class of termination detection algorithms for distributed computations. In: Fifth Conference on Foundations of Software Technology and Theoretical Computer Science, pp. 73–100.

Lai, T., 1986. Termination detection for dynamically distributed systems with non-first-in-first-out communication. Journal of Parallel and Distributed Computing 3, 577–599.

Lai, T., Tseng, Y., and Dong, X., 1992. A more efficient message-optimal algorithm for distributed termination detection. In: Proceedings of the Fourth IEEE Symposium on Parallel and Distributed Processing, pp. 274–281.

Lai, T., Wu, L., 1995. An $(n-1)$-resilient algorithm for distributed termination detection. IEEE Trans. Parallel and Distributed Systems 6 (1), 63–78.

Lamport, L., 1978. Time, clocks, and the ordering of events in a distributed system. Commun. ACM 21 (7), 558–565.

Lamport, L., Shostak, R., Pease, M., 1982. The Byzantine generals problem. ACM Trans. Programming Languages and Systems 4 (3), 382–401.

Mattern, F., 1987a. Algorithms for distributed temination detection. Distributed Computing 2 (4), 161–175.

Mattern, F., 1987b. Experience with a new distributed termination detection algorithm. In: Proceedings of the Second International Workshop on Distributed Algorithms, pp. 127–143.

Mattern, F., 1989. Global quiescence detection based on credit distribution and recovery. Inform. Process. Lett. 30 (4), 195–200.

Mayo, J., Kearns, P., 1994. Distributed termination detection with roughly synchronized clocks. Inform. Process. Lett. 52 (2), 105–108.

Misra, J., 1983. Detecting termination of distributed computations using markers. In: Proceedings of The Second Annual ACM Symposium on Principles of Distributed Computing, pp. 290–294.

Misra, J., Chandy, K.M., 1982. Termination detection of diffusing computations in communicating sequential processes. ACM Trans. Programming Languages and Systems 4 (1), 37–43.

Muller, H., 1987. High level petri nets and distributed termination. In: Concurrency and Nets: Advances in Petri Nets. Springer, New York.

Rana, S.P., 1983. A distributed solution to the distributed termination problem. Inform. Process. Lett. 17 (1), 43–46.

Rokusawa, K., Iciyoshi, N., Chikayama, T., Nakashima, H., 1988. An efficient termination detection and abortion algorithm for distributed processing systems. In: Proceedings of the International Conference on Parallel Processing, pp. 18–22.

Sanders, B.A., 1987. A method for the construction of probe-based termination detection algorithms. In: Proceedings of IFIP Conference on Distributed Processing, 1987, pp. 249–257.

Shavit, N., Francez, N., 1986. A new approach to the detection of locally indicative stability. In: Kott, L. (Ed.), International Colloquium on Automata, Languages, and Programming, Lecture Notes in Computer Science, vol. 226. Springer, New York, pp. 344–358.

Skyum, S., Eriksen, O., 1986. Symmetric distributed termination. In: The Book of L. Springer, New York.

Szymanski, B., Shi, Y., Prywes, N.S., 1985a. Terminating iterative solution of simultaneous equations in distributed message passing systems. In: Proceedings of the Fourth Annual ACM Symposium on the Principles of Distributed Computing, pp. 287–292.

Szymanski, B., Shi, Y., Prywes, N.S., 1985b. Synchronized distributed termination. IEEE Trans. Software Eng. 11 (10), 1136–1140.

Tel, G., 1994. Introduction to Distributed Algorithms. Cambridge Univ. Press, Cambridge, pp. 264–302.

Topor, R.W., 1984. Termination detection for distributed computations. Inform. Process. Lett. 18 (1), 33–36.

Tseng, Y., 1995. Detecting termination by weight-throwing in a faulty distributed system. Journal of Parallel and Distributed Computing 25, 7–15.

Venkatesan, S., 1989. Reliable protocols for distributed termination detection. IEEE Trans. Reliability 38 (1), 103–110.

Ye, X., Keane, J.A., 1991. A distributed termination detection scheme, Tech. Rep. UMCS-91-3-1, University of Manchester, Manchester, England.

Jeff Matocha received the M.S. degree in computer science from Louisiana Tech University in 1995 and the B.S. degree in computer science from The University of Central Arkansas in 1992. He is currently working towards the Ph.D. degree in the computer science department at The University of Alabama. His current research interests include distributed systems, mobile computing, and group communication within the mobile environment.

Tracy Camp received a B.A. degree in Mathematics from Kalamazoo College in 1987, an M.S. degree in Computer Science from Michigan State University in 1989, and a Ph.D. degree in Computer Science from The College of William and Mary in 1993. Currently, Dr. Camp is an assistant professor in the Department of Computer Science at The University of Alabama. Her research interests include distributed systems, network architectures, group communication, and mobile computing. Dr. Camp is a member of both the Association for Computing Machinery and the Institute of Electrical and Electronics Engineers.