# MOBILE AGENTS DEADLOCK DETECTION IN ABSENCE OF PRIORITIES

By
Amr Elkady

A thesis
submitted to the School of Computer Science
and the Faculty of Graduate Studies and Research
in partial fulfilment of the requirements
for the degree of
Master of Computer Science

Supervisor: Dr. Nicola Santoro, Ph.D.

Ottawa-Carleton Institute for Computer Science
School of Computer Science
Carleton University
Ottawa, Ontario

November 21, 2005

*Your file Votre référence*
*ISBN: 0-494-13456-9*
*Our file Notre référence*
*ISBN: 0-494-13456-9*

# Abstract

The goal of this thesis is to introduce a new solution to detecting and resolving deadlocked mobile agents and study the properties related to detecting a deadlock in simple resource model environment.

To do that, we will consider the problem of deadlocks and the additional complications created by the nature of mobile agents. We will also discuss the traditional solutions for deadlock detection and finally we will analyze the currently known solution for detecting and resolving deadlocked mobile agents.

# Acknowledgements

I would like to thank Dr. Nicola Santoro, my supervisor, to whom I am grateful for all the understanding and the valuable support that he showed towards me during the course of my graduate studies at Carleton University. In addition, I would like to thank him for his able guidance, constructive criticism, useful discussions, brilliant ideas and comments that all together helped in bringing this thesis to reality.

I would like to thank the members of my committee. I would also like to thank Dr. A. Sameh for his support and kindness through the years as well as my friends for their loyal friendship and moral support. Particular gratitude to Carol, Hanene, Dina, & Ildemaro

Most of all I would like to thank my family, especially my aunt who's without her I would have never been at this stage, as well as my parents for their great encouragement throughout my studies. Without their support, this thesis would have not been completed.

*Amr Elkady*

# Contents

# List of Tables

# List of Figures

# Chapter 1

# Introduction

The mobile agents technology introduces new challenges in the distributed computing field. In many distributed applications and algorithms there exist a strong relation between the code and data which leads to various assumptions that may no longer be valid when mobile agents are involved.

Distributed deadlock detection is one of these challenges mentioned when mobile agents are involved. Traditional distributed algorithms make various assumptions in relation to the ordering of messages and the location and the ability to locate both the code and the data (namely the process and resources). In addition, they usually assume that the messages will make it to its destination safely and in the order sent.

Mobile agents as they roam in the network move from one host to the other seeking the resources they need. In addition, their speed of migration varies due to various factors including the processing power of the host they recently migrated to and the

network congestion at the new location. Mobile agents are also prone to failures since they are faced with various possibilities to fail including faulty environments or hosts that go down or get rebooted while hosting this agent.

Traditional distributed deadlock detection techniques runs short when it comes to detecting deadlocked mobile agents. When mobile agents become deadlocked due to the creation of a circular resource/agent dependency chain they may not be reachable directly because their current location may be unknown to the given host. Traditional techniques rely on the assumption that the both the resource and the locking process remain stationary where their location is known. As a result, when mobile agents are involved, the mobile agents themselves need to take active role in initiating deadlock detection cycles as well as taking part in coordinating resource usage and making themselves reachable in some cases such that a deadlock situation can be resolved.

Today, there are few published algorithms that are referred to as mobile agents deadlock detection algorithms. However, these algorithms only utilize mobile agents or mobile-agents-like probes to detect deadlocks of static processes and resources. This categorizes these algorithms as traditional distributed deadlock detection algorithms as they do not address the issues related to locking process's mobility for example. In addition, these algorithms continue to be based on assumptions that are not suitable for resource-consuming mobile agents (also referred to as consumer agents) . Examples of such assumptions include message ordering and error-free communications.

There exists only one known solution that specifically addresses deadlocked resource-consuming mobile agents. This solution presented by Ashfield *et al.* [2, 3], as "Deadlock Shadow Agent Approach" (SA) specifically addresses distributed deadlock detection issues related to deadlocked consumer mobile agents including mobility, fault tolerance, and the inability to assume message ordering for deadlock detection. This algorithm however did not totally escape some of the traditional assumptions; for example: assuming global pre-set unique priorities for consumer agents in order to resolve deadlocks in a fashion similar to traditional process based distributed deadlock detection and resolution algorithms. In addition, this technique although will successfully detect and resolve deadlocked mobile agents can prove to be an overkill to utilize in terms of processing power and resource utilization. In particular, it can be unbounded in absence of deadlocks. The deadlock shadow agent approach is described in more details in a later chapter.

## 1.1 Framework

This thesis is intended to study and provide a new solution for detecting and resolving deadlocked mobile agents. Within this framework, mobile agents can lock resources and then move within the network to new location.

In doing so, this work is presented for the single-resource model of deadlocks as described in sections 2.2.2 and 5.1.1 and with basic assumptions of two-phase-locking, error-free communication, and no spontaneous aborts of blocked mobile agents.

## 1.2 Contributions

In this thesis we will introduce a new technique to detect and resolve deadlocked consumer agents in addition to partially avoid the formation of new ones. We will mainly emphasize the following properties in designing this solution:

- Bounded cost solution that is capable of detecting if there is or there is not a deadlock.
- Equal consumer agent rights (i.e., no consumer agent priorities)
- Optimized resource usage and detector agent migration
- Total absence of global coordination protocols or entities that can attempt to oversee the correctness of the algorithm

In doing so, the proposed algorithm is inspired by edge chasing algorithms as described in later chapters where we do not attempt to create a global view of the blocked agents but rather rely on a detector agent detecting a cycle as it visits a consumer agent twice. In addition, as a deadlock detection algorithm executes, it creates a list of known deadlock-causing resources to avoid in order to prevent the creation of a new deadlock. This list remains a partial list and is intended to improve performance by minimizing deadlock occurrences though not totally preventing them.

## 1.3 Organization

Chapter 2 of the thesis is dedicated to providing background and necessary overview of mobile agents and deadlocks. This chapter provides definitions of mobile agents, their properties, advantages and disadvantages in addition to issues related to their implementation and usage. In addition, in Chapter 2 we discuss deadlocks and provides the properties necessary for their existence. This chapter also highlights some misinterpretations to what is a deadlock, provides a classification for models of deadlocks, describes various techniques to handle deadlocks, types of distributed algorithms used to detect and resolve these deadlocks, and finally defines the various types of false deadlocks that could be wrongfully detected by deadlock detection algorithms.

Chapter 3 contains the analysis of the currently known mobile agents deadlock detection algorithm namely "Deadlock Shadow Agent Approach (SA)". In this chapter we analyse the algorithm, highlight its properties as well as its advantages and shortcomings. Chapter 4 of this thesis is dedicated to describing the proposed solution and explaining how it is implemented. In addition, this chapter provides a formal description pseudo code and an example of this algorithm's typical execution.

Finally, Chapter 5 is dedicated to providing an analysis of the propsed solution in terms of correctness, complexity, and comparison with previous solution while the final Chapter 6 provides a summary of main contributions submitted in this work.

# Chapter 2

# Mobile Agents and Deadlocks

## 2.1 Mobile Agents

Networked and mobile computing has became a basic everyday technology particularly with the growth of broadband access and the usage of mobile telephony. Today we are using various mobile applications so transparently that we often do not think about it. Applications like web based services and internet shopping have become an everyday task for many people around the globe as access to broadband internet access, wireless networks, and wireless devices has been almost doubling yearly (it is estimated that global sales Wi-Fi enabled laptops has moved from about 9 million in 2003 to 35 million in 2005. it is projected by industry analysts that this number will exceed 60 million by the year 2008).

As the booming in networked and mobile society is continuing, so is the need

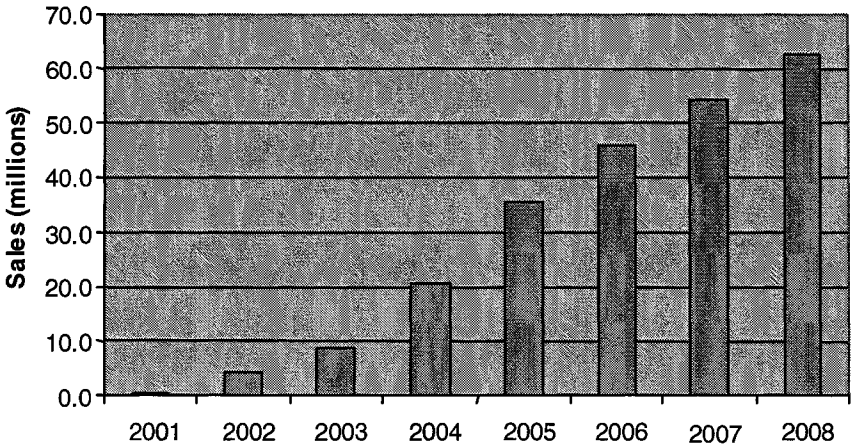


Figure 2.1: Growth in WiFi Device Sales

for applications and services that are capable of coping with the demands of this new world and the need for bandwidth optimization to support the industry's infrastructure and the high demand for bandwidth. In the effort to find more bandwidth efficient schemes and solutions, it has been recognized that the volume of data being processed is becoming larger and larger and with the traditional approaches where code remains static and data is being transmitted to be processed a significant amount of bandwidth and resources are being scarified. An early approach to over come this issue was the use of remote procedure calls which became the foundation of the client-server concept. In this approach, clients send requests to remote code located on a server and get the result back. However this has various drawbacks as it causes many transactions on the network and it is dependant on the network's continuous connectivity.

Another approach that attempts to solve the issues with client-server computing is the mobile agent technology.This technology attempts to remove the notion of clients and servers and as a result minimizes or eliminates the request/response mechanism of

the client-server architecture saving significant bandwidth which is a valuable resource for distributed applications [51, 58]. the commonly it is defined as "An independent program which executes on behalf of a user and which moves in the network to perform its function"[58].

### 2.1.1 Mobility

*Process migration* is the term traditionally used to describe the concept of moving processes from one place to the other [49]. This term has existed for as long as distributed computing has and it refers to systems where processes are moved due to the distributed operating system's decision to move a process from one location to another for reasons that may include fault tolerance, load balancing, or parallel processing. In other words, processes in such systems do not move due to their own desire to move but due to the system's decision to do so. On the other hand, with *Mobility* the when and where decision to move is done by mobile agents and executed via the hosting environment.

#### Evolution Of Mobility

The Evolution path for mobility has started with the networked computing. As computers became interconnected it created the desire to move entities and objects between computers. First, it was the movement of simple data which led to moving the execution control along with the code and the execution environment. In this way mobility has evolved in the following main steps [6]:

1. *Mobility of files:* Where files can be transferred from one machine to another for processing. For example FTP protocol.

2. *Remote procedure calls (RPC):* Remote procedure calls extends the execution control to another machine in order to evaluate and process the data sent to the remote machine. RPCs are in essence a function call where the function resides on a remote machine and where the calling function waits for the result to be returned from this function call.

3. *Remote evaluation(RE)/code on demand (COD):* As an evolution to RPCs, came the idea of moving code to remote machine. In this case, code is sent to a remote machine for execution. An example of this evolution is HTML based code.

Cabri *et al.*[6] summarizes the different evolutions of mobility in the following table:

| **Mobility** | **Local Host** | **Remote Host** |
|---|---|---|
| RPC | control | processor, code |
| Remote evaluation | code | processor, resources |
| Code on demand | processor, resources | code |

Table 2.1: Different evolutions of mobility

### Types Of Mobility

Today's mobile objects are becoming more complex than a few lines of code or few kilobytes of data. Both active objects and mobile agents are a good example of such objects and when considering the mobility of such complex objects, we must take into

consideration that we need to consider their execution state and data stack. In doing so, two kinds of mobility have to be distinguished:

- *Weak:* Weak mobility allows the migration of both code and the values of its variables. After migration, the data and variable values are preserved while the agent is restarted which causes the execution to start from the beginning or from a specific method. Implementation of weak mobility is possible in Java by means of object serialization using either sockets or RMI.

- *Strong:* In that case, both the code and the entire execution state of the object are being moved allowing the execution to resume from exactly the same point where is was stopped before migration. Implementation of strong mobility using languages like Java would require special modifications to allow for the the ability to extract and serialize the stack and execution counter along with the object [6].

### 2.1.2 Definitions

Different authors and researchers in the agent technology have proposed many definitions for what is an *"Agent"*. Some of these proposed definitions are as follow:

> "What I take to be the characteristics of a mobile agent system include
>
> - Use of active objects-that is, objects that move from place to place and, when they arrive at a place, can obtain a thread of control automatically;

- Interaction with various environments to find out information or perform actions on those machines;
- Ability to obtain information or performs actions on behalf of the person who sent the agent out on the network;
- Ability to, after doing its work, report the work's results to the requesting person; and
- Ability to make decisions on behalf of the person it represents."

(J. Waldo, [55]).

"Mobile agents are actually programs that can move from one host to another. They can initiate their own transfer by executing a special instruction in their code " (Hong Wang *et al.*, [56]).

"A mobile agent is a program that can migrate from one networked computer to another while executing " (William Farmer *et al.*, [17]).

"A mobile agent consists of a self-contained piece of software that can migrate and execute on different machines in a dynamic networked environment, and that senses and (re)acts autonomously and pro-actively in this environment to realize a set of goals or tasks" (Ravi Jain *et al.*, [31]).

"An independent program which executes on behalf of a user and which moves in the network to perform its function" (D. Wong, [58]).

"A Mobile Agent is specialized in that in addition to being an independent program executing on behalf of a network user, it can travel to multiple locations in the network. As it travels, it performs work on behalf of the user, such as collecting information or delivering requests" [29].

"An intelligent mobile agent is a computer program which can independently or semi-independently perform tasks which the operator could not perform on his own because of the time or effort required for the task" (Frew *et al.*, [20]).

"Mobile Agents are computer programs that act autonomously on behalf of the user, and travel through a network of heterogeneous machines" (Pleisch *et al.*, [46]).

"Mobile Agents are programs with persistent identity which move around a network on their own volition and can communicate with their environment and with other agents" (M. Jazayeri *et al.*, [32]).

"Mobile agent is a computer program that acts autonomously (independently) on behalf of a person or organization. It is not bound to the system where it begins execution. It has the unique ability to transport itself from one system in a network to another. When a mobile agent moves or is moved to another place, its code and execution states also move with it. After it reaches a new place, the mobile agent will restart execution from where it stopped with the execution states it had at moving. The

re-execution time point is determined by the functionality of that agent" (Dianlong Zhang, [59]).

"A mobile agent is an autonomous program that can move from machine to machine in a heterogeneous network under its own control. It can suspend its execution at any point, transport itself to a new machine, and resume execution on the new machine from the point at which it left off. On each machine, it interacts with service agents and other resources to accomplish its task, returning to its home site with a final result when that task is finished."(David Kotz *et al.*, [35]).

### 2.1.3 Characteristics Of Mobile Agents

As described in previous section, it is obvious that there is not one agreed upon definition of what is a mobile agent, However, the following properties are an acceptable as a common ground for the main *'philosophy'* [12, 40]:

- Autonomy: The agent keeps the knowledge of internal state as it migrates, and it is in control of its actions including decisions to where and when to migrate.
- Sociality: The agent is capable on interacting with other parties in the system and exchange information and updates. These parties include:
  - Local environment: In order to avoid the problems of the client/server associated with data transfer, Mobile agents move to new local environment in order to process data and access resources. In doing so, they require to

interact with the local environment to allocate resources and accomplish their task.

  - Other agents: Mobile agents need to communicate among each other in order to share knowledge and information that would allow for coordination and decision making between agents. almost all mobile agents platforms offers means of communication between various mobile agents although there is the issue of mobile agents being able to agree to *'meet'* due to their mobility.

  - Users: Mobile agents need to communicate their results back to the initiating users when their task is completed. In some cases this is being done via a dedicated agent at the user's environment that receives the information from other mobile agents and handles the presentation task for the user

- Reactivity: The agent may act in response to environment changes.

- Proactivity: The agent pursues its goal when there is a possibility and opportunity to do so.

- Migration: Migration is one of the main distinguishing characteristics of mobile agents as their code does not require to be executed to the end on one single host. In addition, due to the mobile agents' autonomy characteristics, they have control on the decision of when to move and where.

- Data Acquisition: Mobile agents interact with other parties in the system to acquire information in order to achieve their tasks. This acquired information gets processed by the mobile agent or it could be carried to be forwarded to

another entity. The mobile agents may also carry the resulting information resulting from processing data in a given local environment.

- Route Determination: Mobile agents' decision to migrate is mostly driven by the need to acquire further data or resources. The decision of where to migrate next is derived by one of three methods:

  - Predetermination: In this case, the mobile agent is given a travel itinerary to follow when the agent is launched. This approach is used in many cases when there is a need to carefully plan resource usage or when there is a need to plan the order or nodes visited due to given dependencies outside the mobile agent's scope of decision making.

  - Dynamic determination: This method provides the mobile agent with complete freedom of choice as where to go next. The decision to choose a node can be based on the mobile agent's acquired knowledge as a result of its own information processing or via knowledge sharing with other agents in the system etc. It can also be a random decision at times when not much information is known to where is the best node to migrate to. The dynamic determination approach is particularly useful in large networks environments and in agents who are mostly concerned with data mining like web indexing agents for example.

  - Hybrid determination: As the name implies, this method is a hybrid of the two previously discussed methods. In this case, the agent is provided with a visit (or avoid) list of nodes that it must use but it is also given the freedom to choose extra nodes to visit and interact with in addition to the

ones in this list. In other cases, the agent is provided with a site criteria to qualify the sites it can visit and is left to determine based on that which nodes to visit and which to avoid. This method is rather popular in cases when quality and trustworthiness of data are of importance.

### 2.1.4 Advantages

Mobile agents, like other technologies offers or promises to offer various advantages. For example, the notion of moving the code to where data is located can prove beneficial and more efficient in applications like searching large databases[17]. Harrison et, al [23] identifies a number of potentially desirable reasons for adopting a mobile agent paradigm:

- **Efficiency:** The ability of an agent to move to the location of the resources allows various aspects of bandwidth optimization and resource usage. Agents are now able to process data on the server where data resides and transfer only results back where in traditional solutions all data would have needed to be transferred in order to be processed.

- **Persistence:** Mobile agents roam the network independently without the need for a persistent connection from its launching node and will not be affected by this node's disconnection from the networks for any reason that may include the node's failure. This allows mobile users to connect only to launch their tasks as a mobile agent and then disconnect, connect again to receive their results or check on the progress of their agent.

- **Peer-to-peer communication:** Client-server computation model has a distinct differentiation between what is a client and what is a server. In this model, servers and clients are able to connect and communicate but servers and clients are not able to communicate between each other since servers can not be clients to other servers for example. Mobile agents however do not have such restrictions as they are capable or communicating and interrogating resources in addition to other mobile agents in the system which allows for greater flexibility in designing distributed systems.

- **Fault tolerance:** Since mobile agents do not need to maintain a persistent connection with a server in order to keep their status, it is easier for an agent to tolerate and work around network connections losses since their state is centralized. This is in comparison to the client-server model where the transaction state is shared between the clients and server and it is difficult to reclaim to synchronize when the connection is lost [12, 23].

In addition, though non-functional, mobile agents exhibit some advantages which escape their static counterparts [29, 43, 57, 59]:

- **Performance:** The processing is done on one local computer at a time rather than passing commands and responses back and forth across the network.

- **Customization:** Mobile agents allow the creation and addition of new services and functionalities to extend existing software without the need to replace the existing software.

- **Concurrency:** Sending multiple agents to multiple sites allows the possibility

of simultaneous execution of tasks on different sites.

- **Convenience:** Mobile agents can make the design and development of distributed systems and applications easier as mobile agents are naturally distributed in design. Applications utilize mobile agents uses the network to transparently accomplish their tasks while taking advantage of local resources available at various servers.

- **Creativity:**Mobile agents scheme provides an opportunity for new creative design paradigms as they allow for the autonomic mobile computing which can radically change the conventional design process and allow for some highly innovative products to emerge.

- **Asynchronous computing:** Mobile agents can be launched to do their task while the user is disconnected. At some time later when the user connects back the results gets delivered.

- **One Side Programming:** Unlike the client server paradigm, mobile agents do not require synchronized programming efforts between clients and servers in order to ensure interface and functional compatibility between servers and every client is can connect to. In mobile agents, the programming and features are encapsulated within the agent which reduced the programming efforts as well as eliminates the need for revision control.

- **No need for persistent connection session:** Since mobile agents execute their tasks locally and autonomously on the remote host, a network connection is needed only for a brief time in order for the mobile agent to migrate to the

new host. After that point, all the transactions are done locally and the network connection can be terminated.

- **Reduction of network traffic:**As previously discussed, mobile agents' transactions are done locally on the host environment. This as a result does not require passing the calls and results back and forth between a client and a server. As a result network traffic is largely reduced as only two transactions are needed one to migrate the agent to a new host and another to migrate it back.

- **Reduced communication costs:** As a result of the reduction of network traffic and the elimination of the need for persistent connections, mobile agents provide const saving measure for metered network connections (mobile applications for cellular phones are a good example) as both the connection time is reduced as well as the bandwidth consumption. For example, it is more efficient to have an agent sent to a server, do the search, pick the image the user wants and return rather than having to download all the images for the user to browse through to choose the one user wants.

In their conclusions, Harrison *et al.* [23]. argue that, mobile agents' advantages for application can only be seen when network latency is an issue for real time applications. Other than that, anything that can be done with mobile agents can also be done with other existing technologies. However, they conclude that the advantages in mobile agents are not in the individual advantages they offer but in the aggregate set of advantages altogether [12].

### 2.1.5 Typical Problems And Related Work

Mobile agents are becoming a programming and computational paradigms and are being intensively investigated today in the Software Verification community and the Algorithms community. Problems of particular interest include *Leader Election and Rendezvous* .

#### Leader Election

Choosing a leader in a network, or identifying a single object in a group of different objects, has been one of the most analysed topics in Distributed Computing.

The *election* process (also called *symmetry breaking*) is a basic computation in distributed environments. Electing a leader in a distributed system is important when some centralised coordination is required in the environment, either because there is not an available technique to solve a problem in a completely distributed way, or because a centralised approach provides a better solution. Since election provides a mechanism for breaking the symmetry among the entities in a distributed environment, it is at the basis of most control and coordination processes *(e.g., mutual exclusion, synchronisation, concurrency control, etc.)* employed in distributed systems, and it is closely related to other basic computations *(e.g., minimum finding, spanning-tree construction, traversal, etc.)* [44].

The *Election* problem requires one agent to enter the distinguished state of being a *leader* , while all other agents enter another distinguished state, called *follower*, within finite time[13]. A *leader* is recognized by all other *followers* as distinguished

to perform some special task. The *leader election problem* is the problem of choosing a leader from a set of candidates, given that every entity is only aware of its own identification, *i.e.,* starting from a configuration where all entities are in the same state, they arrive at a configuration where exactly only one entity is in state *elected* and all other entities are in state *defeated* [53].

### Rendezvous

In general, the *rendezvous* problem is similar to and includes many situations from real life. In human life, people need to gather and meet in order to socialize, discuss, accomplish tasks etc. To do so, they need to find a way to mutually know where and when to meet which can be complex task in a distributed environment.

The *rendezvous* problem has been extensively studied in the operational research field. However, the computer science field has also been interested in the *rendezvous* problem as a search and optimization problem . In relation to mobile agents, the research in this problem is concerned with the need of a number of autonomous agents to *rendezvous* at a given node while optimizing their steps to do so and/or their memory and resource usage. In addition the issue of symmetry, whether it is related to topology, structure, agents & agents' identity, algorithms is more likely in mobile agents environment and is important to be taken into account when studying *rendezvous* of mobile agents [18, 21].

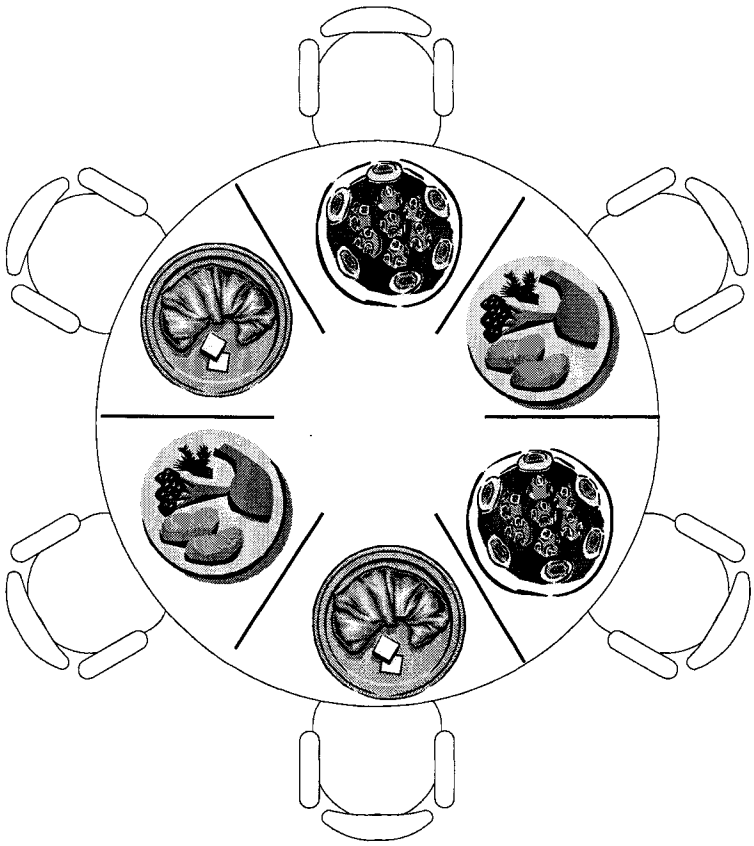

Figure 2.2: The Dining Philosophers

## 2.2 Deadlocks

The deadlock is a situation that can occur when a set of two or more processes are competing to gain exclusive control over a finite set of resources. A Deadlock can be defined in simple terms as follows: *"A set of processes is deadlocked if each process in the set is waiting for an event that only another process in the set can cause"*[4].

Deadlock situation can be illustrated using the classic dining philosophers example. The story goes that a set of oriental philosophers are spending their time either

thinking or eating. When they sit to eat, they are sitting on a round table with a chopstick between each of them as in Figure 2.2. In order for the philosophers to eat, each philosopher need two chopsticks namely the one to the left and the one to the right. A deadlock situation occurs when each philosopher grabs the chopstick to the right which causes the situation where each philosopher has one chopstick and waiting for another one to be freed. Hence, the wait will continue forever as long as the situation remains the same. This is an example of a deadlock [30].

For a deadlock to occur, a set of four conditions need to apply as classified by Coffman *et al.* (1971) [10] and illustrated by A. Tanenbaum [52] where if any of these four conditions is absent a deadlock can not occur in the system[10, 52]:

1. **Mutual exclusion:** Each resource can be used by only one process at a time.
2. **Hold and wait:** A process currently holding at least one resource can keep holding them and request and wait for additional resources that might be held by other processes.
3. **No preemption:** A resource can be forcibly taken away from a process but voluntarily and explicitly by the process holding it.
4. **Circular wait:** There exists a circular chain where a set of two or more waiting processes are waiting for a resource held by the next process in this chain. For example, if there exists a set of processes $P_0$, $P_1$,..., $P_n$ such that $P_0$ is waiting for a resource that is held by $P_1$, $P_1$ is waiting for a resource that is held by $P_2$,..., $P_{n-1}$ is waiting for a resource that is held by $P_n$, and $P_n$ is waiting for a resource that is held by $P_0$

The first three conditions are considered policy conditions. A deadlock occurs only if the circular wait condition exists and is unsolvable due to the existence of the three policy conditions.

That being said, it should be mentioned that caution should be taken when illustrating a deadlock example that the previous conditions are properly utilized. In a more recent deadlock classification work by G. Levine (2003) [39], Levine states that although many of the operating systems textbooks provide and describe the above mentioned characteristics in great details, they fail to provide correct examples. The texts of Silberschatz *et al.*[48] and Stallings [50] among others are used as an illustration for incorrect deadlock examples. Levin notices [39]:

> *Silberschatz* et al. *claim that a deadlock exists if two trains on different tracks approach a crossing and wait for the other to proceed. {Yet neither train is holding a resource requested by the other.} Similarly, Stallings gives an example of four cars at different stop signs approaching an intersection, where each gives precedence to the car on its right. {Yet none of the cars (or trains) have been allocated intersection , which are the only resources that are mutually requested. Nor is roll back required to break the wait.}* "

### 2.2.1 Representing Deadlocks

In order to visually represent deadlock, a global representation of the system can be modeled by means of a Wait-For-Graph (WFG). WFG is generally a directed graph

with nodes representing processes and edges that represent waiting states as a result of a sent requests that has not received a reply [5].

Holt (1972)[27] represented a method to model the deadlock conditions utilizing the directed graph where nodes are represented by two types: circles that represent processes and squares that represent resources. In this graph, edges either represent a transaction waiting for a reply if they are originating from a process (represented by a circle) or, if the edge is outgoing from a resource (a square), they represent a resources that was granted to a given process as a result to a previous request[27, 52].

We will be utilizing similar representation in this work to model the mobile agent waiting and deadlock situations. We will however represent the resource ownership via non-directed edges to ease the illustration. In addition, we will utilize the notation
(A[$R_3$ , $R_6$ , $R_8$], D) $\rightarrow$ B[$R_1$, $R_7$] $\rightarrow$ C[$R_2$] $\rightarrow$ A
where in this notation, agents A (that owns resources $R_3$ , $R_6$ , $R_8$) and D are both waiting for agent B (that owns resources $R_1$, $R_7$). Agent B at the same time is waiting for agent C (that owns $R_2$) which in turn is waiting for agent A. This notation sample provides an example of a deadlock situation.

### 2.2.2 Models Of Deadlocks

The system structure may allow additional complexities in the deadlock problem. Depending on the application, systems allow for a various kinds of resource requests. The complexity of the model of requests allowed need to be taken into consideration by deadlock detection algorithms as it characterizes the occurrence of deadlocks. A

survey by Knapp [33] defines a set of models including:

- Single resource model
- AND model
- OR model
- AND-OR model

### Single Resource Model

In the single resource model, a transaction can request only one resource at a time. As a result, there is only 1 possible outer edge of the WFG at a time. This model is widely used in theoretical studies of database systems where finding a deadlock corresponds to finding a cycle in the WFG [33]

### AND Model

Chandy and Misra[7] provide a formal definition to the AND model. In this model, a transaction is permitted to request many resources at the same time and block waiting for all these resources to be granted to it [30] hence the name AND model. As a result of this resource model, the WFG would have more than one outgoing edge at a time. Finding a deadlock in this model is still is based on finding cycles in the WFG [33] although there is the possibility of overlapping cycles which would increase the complexity of algorithms in order to avoid false deadlock resolution.

### OR Model

In the OR model, the request to obtain various resources can be satisfied by having any of them granted. An example for this would be a read request for a replicated data file, in that case granting any of the file's instances would satisfy the request. In this model, the discovery of a cycle in insufficient to deduct that a deadlock exists as any of the processes involved in this cycle may have another non-cycled dependant edge which would allow this process to gain the resource it needs and abandon the request involved in creating this cycle. As a result, detecting a deadlock in the OR models involves finding a *knot* in the WFG where no paths originating from the center of this knot have dependants. Finding this *knot* signifies that all outgoing edges are a dependant set. It is worth mentioning however that a process could be deadlocked by depending on a deadlocked process [30, 33].

### AND-OR Model

The AND-OR model can be seen as a generalization and combination of the AND and OR models where requests of both types are allowed. For example, a process may request resource $a$ and either $b$ or $c$. In this case if it received $b$ first it must cancel its request for $c$ but continues to wait for $a$. This model is particulary advantageous in models involving arbitrary resource demands involving AND and OR functionality [8, 30, 33]. An example of utilizing this model can be applied using the dining philosophers where each philosopher needs a right AND left chopstick whoever, the same philosopher can get ANY right chopstick and ANY left chopstick.

The AND-OR model can be detected in principle by repeatedly applying the methods used for the OR model detection. More advanced work however was done by Gray *et al.* [22] and Herman *et al.* [24] in addition to Jain *et al*'s new proposal [30].

### 2.2.3 Deadlock Handling

Deadlocks can be dealt with by one or more of four methods: we can choose to ignore the deadlock altogether *"maybe if you ignore it it will ignore you"* [52]. Or, we can prevent the creation of a deadlock by building a policy that prevents one or more of the necessary conditions to occur. Third, we can avoid the formation of a deadlock by allowing locks based on the current allocation state. Finally, we can attempt to detect and recover from deadlocks [47, 52].

#### Ignoring Deadlocks

Also referred to as the Ostrich algorithm where the solution is to stick one's head in the sand and pretend that there is no problem[1]. The feasibility of this approach varies by application and need vs. availability of both resources and processes. For many operating systems including UNIX and Microsoft Windows for example, this approach is in many cases preferred as it is assumed that users would rather have an occasional deadlock than being restricted in their usage and availability of resources. This can be seen as a tradeoff between convenience versus correctness due to the cost

[1]We should mention however that Tanenbaum [52] strongly defends ostriches in a footnote of his book.

of deadlock prevention and/or recovery [52].

### Deadlock Prevention

The basic strategy for deadlock prevention aims towards ensuring that at least one of the policy conditions for deadlocks will be prevented from occurring or to prevent the possibility for an unsolvable circular wait. With one of these basic deadlock conditions missing, a deadlock can not be formed[28, 52]. Tanenbaum [52] and Shub [47] however conclude that attacking the first (mutual exclusion) and third (no preemption) conditions are not practical approaches leaving the hold-and-wait and the circular-wait as possible conditions to attempt to prevent.

Tanenbaum [52] summarizes the possible approaches to deadlock prevention in the following table:

| **Condition** | **Approach** |
|---|---|
| Mutual exclusion | Spool everything |
| Hold and wait | Request all resources initially |
| No preemption | Take resources away |
| Circular wait | Order resources numerically |

Table 2.2: Summary of approaches to deadlock prevention

Shub [47] in the attempt to suggest more solutions for the hold-and-wait problem, suggests resource ordering as a way to eliminate hold-ad-wait condition. This alone as a matter of fact only eliminates the circular-wait condition by forcing processes to request resources in order but not the hold-and-wait condition since processes will continue to hold resources and request and wait for more. However, utilizing resources pre-allocation combination will effectively negate the hold-and-wait condition[39].

### Deadlock Avoidance

The differences between deadlock prevention and deadlock avoidance are considered insignificant in many cases [41]. The main differentiating factor between the two schemes is that for deadlock avoidance, the main purpose is *"to avoid deadlock by using information about the process' future intentions regarding resource requirements"*[28].

Deadlock avoidance algorithms rely on defining a *"Safe State"* where there exists an ordering of processes where each process in turn can obtain the resources it needs (within a predefined limit) and run to completion. Resources are requested one at a time and granted only if the system will remain in a safe state after the request is granted where all processes will have at least one way to complete execution [28, 47, 52].

### Deadlock Detection and Recovery

Systems that allow for transactions to wait freely for resources in a conflicting lock situation are typically the ones that utilizes the deadlock detection and recovery schemes. These systems are typically considered to adopt a very liberal resource allocation policy in relation to the ones utilizing deadlock prevention or avoidance techniques as they assume all resource requests will be granted eventually[28].

Deadlock detection schemes rely in essence on detecting and resolving cycles that represent deadlocks in the directed wait-for-graph. If a deadlock is detected, the system attempts to recover as gracefully as possible. The recovery can be done via

killing one of the processes involved or force it to restart, or rolling back the process to a point in time prior to it having the resource and allowing this resource to be assigned to a different process. In many cases however it is possible to take the resource from the process allowing the deadlock to be resolved [16, 28, 52].

In the following table we will attempt to provide a basic comparison between deadlock prevention, detection, and avoidance techniques to highlight their differences, major advantages and shortcomings. The table is based on the comparison done by Isloor and Marsland [28].

| | **Detection** | **Prevention** | **Avoidance** |
|---|---|---|---|
| **Res. Alloc. Policies** | • Very generous<br>• Resource requests are granted when possible | • Conservative<br>• Under-commits resources | Average |
| **Strategies** | Periodically test for deadlocks | • Pre-request all resources<br>• Preemption<br>• Resource ordering | Searches for a safe path |
| **Advantages** | Processes can immediately start executing as they are created | • Suitable for bursting activity type of processes<br>• No preemption<br>• No run-time overhead[a] | No preemption |
| **Disadvantages** | • Possible process delays due to preemption<br>• Possible high processing overhead with some algorithms | • Process start delay<br>• Inefficient with real time apps<br>• Over-preemption | • Possible long process blockage periods<br>• Need for pre-determined resource usage |

[a]prevention is a design time solution

Table 2.3: Summary of deadlock handling schemes

### 2.2.4 Deadlock Coordination

One of the fundamental problems in designing distributed systems is the synchronization of access to shared resources while allowing for high availability. In order to do so, most distributed systems adopt "locking" schemes mainly due to their simplicity. In these schemes, each resource has a unique lock that can be held by only one transaction at a time while other agent requiring this resource will have to either wait, abort, or attempt to preempt the current locking agent. A particularly popular locking protocol is the two-pase locking (2PL) which designates two phases for locking and unlocking of resources:

1. Growing phase: An agent may seek and lock the resources it needs for its task.
2. Shrinking phase: An agent may release the resources it acquired in the previous phase.

For each of these two phases, the agent may not perform other locking or unlocking actions outside the restriction boundaries of the phase it is currently in [34, 42].

2PL is often used for deadlock avoidance and resource coordination where if an agent was unable to lock all the resources it needs during the first phase then it can release these resources and restart the first phase again. This strategy is not generally acceptable however for real-tim systems where more recent studies suggests problematic behaviour in such cases. However, this protocol continues to be one of the simple and efficient ways to coordinate and avoid deadlocks [52, 54].

## 2.3 Distributed Deadlock Detection Algorithms

### 2.3.1 Types Of Algorithms

Deadlock detection attempts to find and resolve actual deadlocks by relying on a Wait-For-Graph that in some cases is built explicitly and analyzed for cycles. As published by the ACM computing surveys, E. Knapp [33] classifies distributed deadlock detection algorithms into four categories:

#### Path-Pushing Algorithms

These algorithms maintain the notion of building a global explicit wait-for-graphs by having each site attempt to do so from the information it has available and sending its local WFG to neighboring sites when deadlock computation is performed. The goal of these algorithms is to allow at least one site to have sufficient information to construct a global WFG view and decide if a deadlock exists or not. Knapp notes: *"One noteworthy point about path-pushing algorithms is that many of them were found to be incorrect, either by not detecting true deadlocks, by discovering phantom deadlocks, or both"[33].*

#### Edge-Chasing

These algorithms do not attempt to construct a global wait-for-graph. Instead, they rely on the knowledge that each of the hosts involved has knowledge of some of their

edges. The processes can issue special messages called probes where a probe message is forwarded to all processes that the receiving process waits for. A deadlock can be declared if a probe message makes its way back to the originating process [11, 36]. These algorithms consist of three steps: initiation, detection, and resolution [11].

### Diffusing Computation

The diffusing computation model is based on sending "Are you blocked?" detection probes to dependent processes and wait for a response. These dependant processes in turn repeat the same for their dependant processes and so on. Eventually, returned responses signals a deadlock at the root process. This type of algorithms can be used for complex deadlock models where edge-chasing algorithms can not be used, otherwise it could be an overkill to be used for single resource and AND model deadlocks [36].

### Global State Detection

Where the relevant parts of the WFG graph are constructed in a globally consistent form by taking "snapshots" of the system state and analyzing it for deadlocks.

Knapp's survey mentions among its conclusions:

> *"The large number of errors in published algorithms addressing the problem of distributed deadlock detection [Bracha and Toueg 1983; Chandy*

*and Misra 1982; Ho and Ramamoorthy 1982; Menasce and Muntz 1979; Obermarck 1982 shows that only rigorous proofs, using as little operational argumentation as possible, suffice to show the correctness of these algorithms. By falling back on well-known and completely general principles like diffusing computations and global state detection, it seems possible to achieve both elegance and correctness, even for more advanced models of resource requests, without introducing unnecessary complexity."*

Generally, deadlock detection algorithms are considered correct iff the following two conditions are fulfilled [36, 37]:

1. **Progress:** Every true deadlock will eventually be detected within a finite amount of time
2. **Safety:** Only true existing deadlocks are detected (no false deadlocks)

### 2.3.2 False Deadlocks

Kshemkalyani and Singhal [37] define three types of false deadlocks that affects the accuracy of deadlock detection algorithms: shadow, phantom, and pseudo deadlocks.

#### Pseudo Deadlock

Pseudo deadlock can be characterized in simple terms as a deadlock that has never existed. This could happen mainly due to algorithm errors where as edges appear and

disappear, they are being accumulated by the detection algorithm which concludes the existence of a deadlock that was never present[37].

### Phantom Deadlock

This type of deadlocks is a deadlock that once existed in the system but no longer exist at the time of detection. This occurs as a result of a deadlock resolution event occurring by more than one agent at a time or as a side effect of resolving a different deadlock[2]. Kshemkalyani and Singhal [37] indicate that phantom deadlocks appear as a result of algorithmic errors where the main ones are as follows:[37]

1. *"Information about outdated dependencies not cleaned properly,"*
2. *"Outdated information is propagated, and"*
3. *"Temporal information which can capture causal relationships is not included in the deadlock detection process."*

### Shadow Deadlock

A Shadow deadlock is a deadlock that is *"detected concurrently with its resolution"*[14]. Kshemkalyani and Singhal [37] declare that *"nothing can be done to avoid shadow deadlocks (except in a restricted model)..... because they occur due to inherent limitations"* [37]. A restricted model with the following properties is identified by [37] as necessary model to avoid shadow agents:

- *Single-request model.*
- *Nodes cannot abort spontaneously.*
- *A single resolution for a detected deadlock.*

*"Since a node can make a single request at a time, deadlocks are isolated, i.e., a node can be part of at most one deadlock. Also, deadlock resolution cannot occur concurrently with detection of the deadlock because resolution occurs only as a result of detection"*

These properties stabilize the deadlocks and ensures that no resolution will occur without a detection process and since agents cannot spontaneously withdraw their request; the deadlock will have to persist until an explicit deadlock resolution event occurs [37].

# Chapter 3

# Problem Description and Model

The problem of detecting deadlocked mobile agents adds extra assumptions and parameters that cause traditional distributed deadlock detection algorithms to run short. This is due to mobility of the agents and the inability to locate them directly which is a heavily-relied-upon issue in most traditional deadlock detection algorithms. Traditional distributed deadlock detection algorithms would require various modifications in order to be able accommodate mobile agents environments.

In the following, we will describe the basic assumptions and model used for current solution (Algorithm SA) and the proposed solution. Both solutions will be discussed in details in the following chapters.

## 3.1 Mobile Agents

A mobile agent is an autonomous computational entity that *"can move from machine to machine in a heterogeneous network under its own control. It can suspend its execution at any point, transport itself to a new machine, and resume execution on the new machine from the point at which it left off. On each machine, it interacts with service agents and other resources to accomplish its task, returning to its home site with a final result when that task is finished."*(David Kotz *et al.*, [35]).

The mobile agents these solutions are concerned with are these computer programs which act autonomously on behalf of the user and travel through the network acquiring resources needed for them to accomplish their tasks. These mobile agents are referred to as consumer agents (CA).

Mobile agents characteristics are defined in section 2.1.3. Of particular importance to this problem are the following capabilities of mobile agents:

- Computation: Mobile agents are computational entities that are capable of performing given tasks and make decisions based on the results.
- Autonomy: Mobile agents can decide on which actions to take and when and where to migrate without the need for instructions from an external controller or from other agents.
- Memory: Mobile agents are capable of storing and retrieving information. These information can be carried with the agents as they roam the network. The size of available memory is bounded by design and is subject to other possible

limitations like the host's memory and space availability.

- Strong Migration: Mobile agents are capable of migrating from one node to the other. When they do, both the data state and the runtime state are preserved and migrated along with the agents' executable code.

There are four possible communication methods for mobile agents to interact with each other: boards, message exchange, tokens, and face-to-face communication [19]. In this thesis, face-to-face communication method is assumed as the method of communication between mobile agents. This means that the only way for information to be exchanged between two agents is by having both agents rendezvous at the same host environment.The communication in that case is both asynchronous and instantaneous.

## 3.2 Network Model

Mobile agents move within a network of possibly heterogeneous nodes - or hosts - where different types of nodes are added and removed dynamically. The Internet and mobile networks can serve as a good example where various types of devices either static or mobile connect and disconnect to and from this network. This environment can be described as a undirected connected graph with a set of nodes and edges. Every node has neighbors which are a subset of all the other nodes in the network. For each node in this set, a bidirectional logical communication link exists with every neighboring node. Further, each node can be uniquely identified within the network via the use of a unique identifier. Asynchronous network model is assumed [4].

The deadlock detection algorithms for mobile agents should be network organization independent where neither the nodes nor the agents need to maintain knowledge about the size or topology of the network. Both traditional and mobile agent deadlock solutions maintain an isolation layer from the knowledge of physical network connections and routing details that take place in order to move between one node and the other. They however differ when it comes to the logical view of the nodes and network topology as many traditional solutions rely on properties of specific topology model to function. Mobile agent based solutions - including ours - on the other hand do not assume such knowledge which allows agents to move at will in a dynamic network where nodes are swapped or new nodes added [2].

In order for mobile agents to perform their tasks, migrate and adapt to heterogeneous environments and platforms, an underlying hosting infrastructure is needed. This is referred to as the hosting environment. This environment is a process running within hosts which allows them to become "agent aware". The hosting environment provides - among other functions - the interface between mobile agents and the hosts, provides agents with the resources they need, and facilitates their migration to neighboring hosts. In addition, it allows mobile agents within the same host to communicate, provides them with resource information, and allows them to acquire or release resources [1].

## 3.3 Resources and Deadlocks

We are interested in systems where consumer agents roam the network and attempt to gain exclusive access to the resources they need. In these systems the possibility of deadlocks exists under certain conditions. A set of consumer agents by definition is deadlocked if each consumer agent in that set is waiting for a resource that only another consumer agent in the same set can release.

In order to acquire and manipulate resources, consumer mobile agents follow the following procedure:

- In order for agents to perform their tasks, they need to *"lock"* the resources they need. Any given resource can be locked by only one agent at a time. Locking a resource requires the ability to communicate and coordinate with the host environment to be granted exclusive access to the resources. The host environment is the only authority capable of granting and denying local resource locking requests as well as unlocking its local resources.

- Agents requesting a lock on a resource can neither spontaneously abort their request nor release the resources they acquired except due to explicit deadlock resolution.

- Agents handle the resources in tow phases. In the first phase, agent goes through resources acquisition and locking (Acquire Phase) for all the resources it will need. Then, after successfully acquiring these resources, it can start the second phase of performing its task and releasing the locks on the resources [52].

These assumptions provide the required basic conditions (mutual exclusion, hold and wait, and no preemption) for a deadlock to be possible to occur in this network as described in chapter 2.

Further, the following resource acquisition assumptions are also made:

**Local resource locking:** In order for a resource to be locked or unlocked by an agent, the agent must be physically present at the same environment as the resource. This is in order to allow the host environment to communicate with and locate the agent during the resource locking/unlocking process, as well as when the agent is blocked waiting for the resource.

**Single resource acquisition:** Consumer agents can only attempt to lock one resource at a time. When the resource is granted, consumer agents can request more.

If a deadlock occurs, this resource acquisition model constitutes a single-resource deadlock model where any given consumer agent can only be waiting for no more than one resource at a time.

# Chapter 4

# Existing Mobile Agents Solutions

The work by B. Ashfield *et al.* [2, 3] is the only work currently found in the literature that specifically addresses deadlocked mobile agents. Other work done in the literature mainly utilizes mobile agents in detecting distributed processes. This work however does not account for the specific properties and complexities introduced by attempting to detect deadlocks in mobile agent environment. These properties include the introduction of the mobility factor, since agents may not be directly reachable and their location may be unknown. Many of the traditional distributed algorithms rely on certain assumptions like message passing, static network topology, and the pre-knowledge of the location of processes and resources. These assumptions are no longer valid when we account for the fact that when mobile agents are considered, clients and servers may move or fail.

The Shadow Agent deadlock detection technique (SA) uses the single-resource request deadlock model though not explicitly indicated in the papers. However, it

is mentioned that *"Shadow agents represent at most one outgoing edge in the global wait-for graph. This is guaranteed by the fact that a single agent can only be blocked on one resource and cannot request more resources or perform other actions while blocked"*[2].

This solution can be categorized as a path-pushing type of distributed deadlock detection algorithms where each agent in the WFG attempts to incrementally build a global wait-for-graph and analyzes it for cycles. If a cycle was found in the global WFG, it indicates the existence of a deadlock. In that case, all agents involved will independently realize the existence of the deadlock.

## 4.1 Algorithm Overview

In order to gather the information needed to build the global view of the wait-for graph, shadow agents spawn detector agents that move to where the resources owned by these shadow agents are located and attempt to contact other agents that are blocked waiting for these resources. The detector agents exchange the deadlock detection information with these blocked agents and return to their originating shadow agents. In other words, the detection agents attempt to incrementally construct the global wait-for graph by moving one level down the graph and exchange information with blocked agents there. These blocked agents in turn send their detector agents one level below to gather information and so on. For example, in Figure 4.1, $A_4$ will send a detector agent to exchange information with $A_5$ and return with the results; it will also send another detector agent to $A_6$ to do similar task. In addition, $A_3$ will

send a detector agent to $A_4$ to exchange information and will also send four other detector agents to do similar task with the other four dependant agents that are blocked waiting for the resources locked by $A_3$.

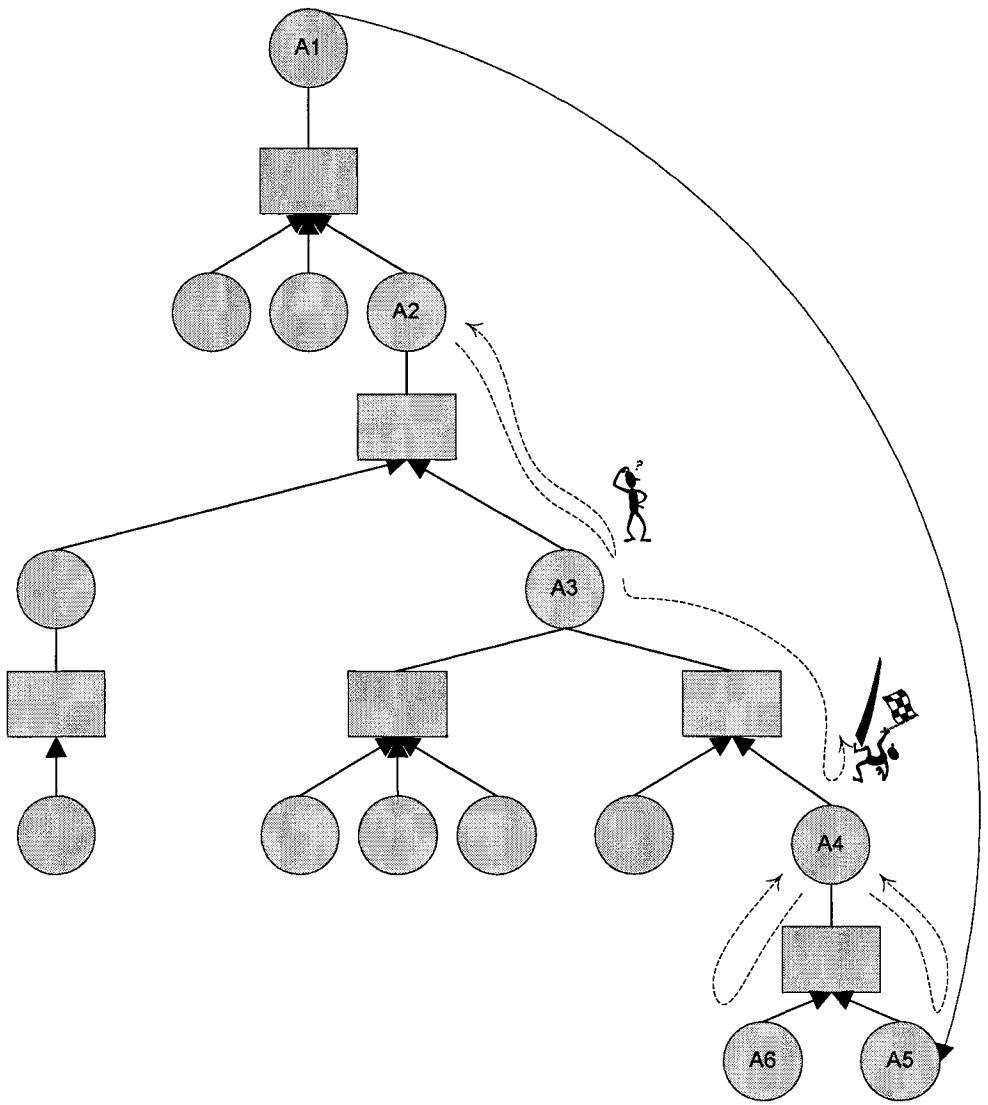


Figure 4.1: Existing deadlock can go undetected due to lack of information

If no deadlock was found in the constructed wait-for-graph, it does not indicate that a deadlock does not actually exist. This un-detection of deadlock is due to

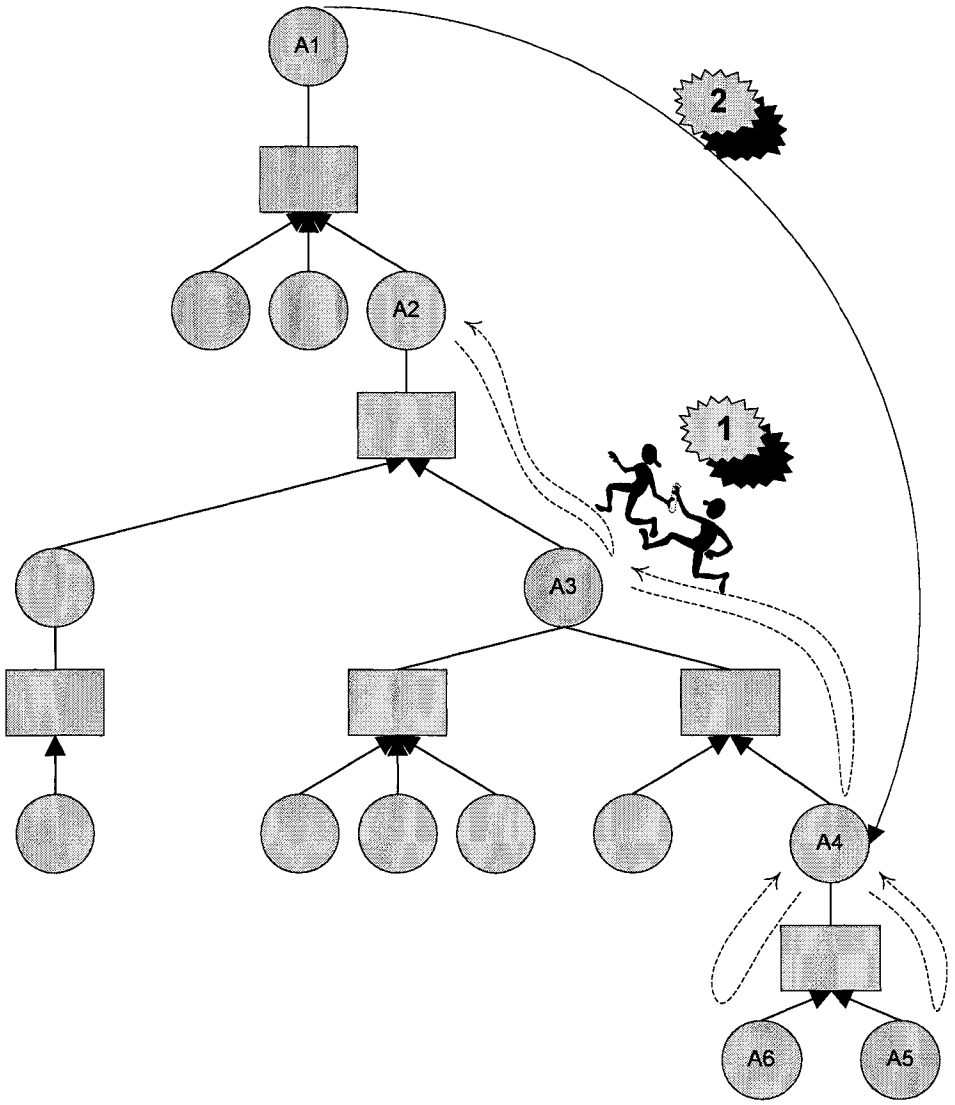


Figure 4.2: Undetected deadlock can form as DA is moving back

various reasons including the possible delay in communicating the information, as illustrated in Figure 4.1, or due to the formation of a new deadlock after the detection agent returned with the information, as illustrated in Figure 4.2. To overcome these problems, the algorithm is forced to continuously repeat until a deadlock is detected or all the resources are released.

In order to collect information about waiting processes, this solution relies on each agent's knowledge of the location of the resources it owns exclusive locks on. By checking with the host environments of these resources, agents can obtain knowledge of dependent processes and start communicating with them and exchanging deadlock detection information in an effort to build the global wait-for-graph.

If a deadlock was detected, all agents that are part of this deadlock will eventually detect the same deadlock by analyzing their own locally constructed copies of the global wait-for-graph and attempt to resolve the deadlock as a result. In order to coordinate the deadlock resolution, algorithm SA relies on pre-assigned priorities for the resources that follow their host environment as detailed later in this chapter. As each shadow agent independently realizes the existence of the deadlock and attempts to start deadlock resolution, each shadow agent searches in its locally constructed wait-for graph for the resource with the least priority and the shadow agent that owns exclusive locks on this resource. This shadow agent is assigned to be the victim agent that must release its resource while other shadow agents do not attempt to do anything. As a result, the deadlock is always solved by only one agent.

The deadlock cycle is broken by having the victim shadow agent send a detector agent to the host environment where the resource that needs to be freed is located. The detector agent informs the host environment to free the given resource and to inform the waiting agents of its availability. The detector agent then returns to its originating victim shadow agent with the success notification so that the shadow agent removes this resource form its list of primary locks.

## 4.2 Unbounded Cost Of Detection

Due to its nature, when a given algorithm cycle is concluded with no detected deadlock, the algorithm is forced to repeat the detection cycle over and over again until one is detected. This repetition is due to the inability of this algorithm to provide a negative confirmation for the existence of a deadlock caused by the possible racing conditions that might occur while exchanging deadlock information between detection agents. This situation can occur when more than two consumer agents exist in the graph as illustrated in Figures 4.1 and 4.2

As a result, in a deadlock-free environment, this algorithm will continue execution infinitely consuming valuable resources.

## 4.3 Agent Identification

Algorithm SA calls for agents to be uniquely identifiable once they lock a resource so that they can be found in the agent system during deadlock detection. The solution however does not go into details on how to create unique identities for those agents. In Ashfield *et al* [3] it is stated that *"The assignment of identifiers may be done before a consumer agent blocks (i.e, during creation) or only once they lock a resource (using a globally unique naming scheme)"*

The idea of not requiring unique ID for the consumer agent until it blocks on a resource is rather unusual. This is largely due to the the algorithm's semantics where there is no need to reference an agent until the agent is actually blocked on a resource.

In that case, it will need to be referenced by the detection agents seeking consumers agents blocked on the resource that their shadow agent owns.

This however, may lead to the assumption that it is possible to create non-uniquely identifiable agents in the system although actually a unique identification is needed from the time of the agent's creation in order for it to be hosted by environments and be routed in its network movements. In order to deal with this issue, algorithm SA [2] uses *"environment tokens"* which are not elaborated much upon but it can be understood from the text that they are created at the time of agent's creation and they allow to uniquely address this given agent.

In Ashfield *et al* [2] it is stated that

> *"Consumer agent monitoring is performed through "environment tokens"; therefore, each time an agent arrives at a host environment it must present its token. This token has no meaning to the agent, and is only used by the host environments to coordinate the deadlock detection process"*

The existence of this token is what allows the host environments to uniquely identify and differentiate between one agent and another though it is being looked at as a system-based identifier. So, in reality, the algorithm still relies on the creation of a unique identification for the agent from the time of its creation. However, this identification is ignored as system-based and another one is later created when an agent locks a resource.

To summarize, algorithm SA requires three different kinds of unique identifiers while claiming there is no need to have one:

1. Environment tokens
2. Globally pre-assigned priority
3. Unique agent identification created in reference to host

## 4.4 Shadow Agent

Algorithm SA requires the creation of a shadow agent. A shadow agent is a clone of the main consumer agent. The shadow agent allocates and keeps track of the resources being held by the consumer agent [2, 3] such that they can be located during deadlock detection. Shadow agent also processes the data gathered by detection agents as well as initiating the deadlock detection in the 1st place. Said in another way, the shadow agent is required to do all the "processing" related to deadlock detection needed by this solution.

That being said, shadow agents are needed mainly in order to handle the processing required by the deadlock detection cycles as they repeat infinitely until a deadlock is detected. Shadow agents are also responsible for locally building the global WFG by analyzing and concatenating the information gathered by detector agents in order to build one coherent WFG. In addition, shadow agents need to analyze the global WFG to find cycles that denote deadlocks. It is mentioned in [2] that:

> *"Upon arrival, the detector agent informs its shadow that it has returned and communicates its constructed deadlock table. The shadow agent analyzes this table, which represents the global wait-for graph, to determine if*

*a deadlock is present"*

That being said, since the shadow agent is a clone of the consumer agent, the existence of a shadow agent per consumer agent doubles the storage requirements and cuts in half the host environments' capability to host consumer agents. In addition, the shadow agent is required to move along with its consumer agent; This fact doubles the load of relocating and routing consumer agents in the network.

## 4.5 The Transaction priorities problem

Algorithm SA relies heavily on preassigned global priorities in order to be able to organize and resolve deadlocks. As described in [2]

> *"Using this technique, when each node is added to the mobile agent system it is assigned a priority identifier. The priority of the resource, and hence any locks on that resource, have the same value as their host environment. In a similar manner to the protocols that must be present to establish routing tables among agent environments, a protocol must be present to negotiate environment priorities. This can be implemented as a continually running and updating protocol, or it can be central authority which must be contacted as part of an agent environment's initialization. Regardless of the implementation, the priority protocol assigns each agent environment or resource a priority that can be used when deciding which resource should be unlocked to resolve a deadlock."*

Hence, this solution relies on the assumption that when a deadlock exists, all shadow agents involved will have unique priority which allows them to know which resource and agent will be the one resolving the deadlock based on it having the least priority.

As it assumes that priorities are assigned per host, Ashfield [2] does not however illustrate on the situation when two nodes are part of a larger deadlock and both of them are on the same host. It is mentioned in another part of the thesis *"If the locking agent is off-site, the distributed deadlock detection sequence is initiated by the shadow agent, otherwise, no special processing occurs"*[2]. This statement seems to imply that the solution may not consider the existence of two or more dependant nodes coexisting in the same host and are part of a larger deadlock that spans several hosts. In that case, if this host happens to have the least priority it arises the question of which agent will be the victim agent responsible for resolving the deadlock. Otherwise, both agents may try to take responsibility for breaking the deadlock causing a phantom deadlock problem.

In all cases, a global unique assignment of priorities does need to exist based on an acceptable priority criteria. This may not be feasible or practical in a large and truly distributed environments such as the internet and the World Wide Web.

## 4.6 Summary

This algorithm provides various advantages and and new aspects in relation to traditional distributed deadlock detection algorithms. That includes:

- It accounts for agent mobility where a consumer agent may not be easily located as it roams the network.
- It accounts for fault tolerance where a detector agent may fail unexpectedly. The fault tolerance technique is mainly based on calculating timeout value for the round trip of the detector agent.
- The creation of a shadow agent frees the deadlock processing tasks from the consumer agent allowing it to continue its normal processing in relation to its original task.

However, applying this algorithm has some disadvantages and shortcomings mainly due to the nature of the algorithm, some of the important ones to note are:

- Significant processing is needed by each shadow agent to reconstruct and anayl-ize the WFG in search for cycles.
- Due to its inability to provide a negative confirmation for the existence of a deadlock, the algorithm has to execute continuously in the absence of a deadlock until a deadlock is formed and detected.

In addition, some shortcomings of the implementation of this algorithm include the following:

- Since the shadow agent is a clone of the consumer agent, the existence of a shadow agent per consumer agent cuts in half each host environment's capability to host further consumer agents. In addition, the shadow agent is required to

move with its consumer agent which doubles the heavy load of moving consumer-like agents.

- The algorithm actually uses two unique identification mechanisms to identify agents. One is a system based "tokens" and the other is an agent-id.

# Chapter 5

# Proposed Solution

A major problem with deadlock detection for mobile agents is the fact they are, as the name implies, mobile. Hence, it is not possible to directly locate and control them by any given host which causes traditional distributed deadlock detection algorithms to fall short. Algorithms for detecting mobile agents' deadlocks need to take into account the fact that a mobile agent will be roaming the network and there may not be any direct way to reach it. In addition, various distributed deadlock detection algorithms rely heavily on the accuracy of messages and their delivery ordering which would require high reliability performance from the network and the host servers. Unfortunately, mobile agents are prone to failures as they move through the network and are hosted by various servers and agent environments.

This solution mainly addresses the mobility problem associated with deadlock detection for a mobile agent system while attempting to be more efficient than currently available solutions.

## 5.1 Solution Properties And Assumptions

Detector agents need to be able to uniquely identify and distinguish each consumer agent they are searching for. As a result, each agent must hold an identity that allows it to be uniquely addressed in the network. By default, any host in a given network carries a unique identity within that network. We will assume that within any given host the created agents will have their own local ID. As a result, a globally unique ID for each agent can be achieved by combining properties like *originating host+time of creation* [1].

Agents need to communicate with the host environment in order to coordinate resource access. The host environment is the one-and-only authority able to approve or deny the agent's requests. If an agent was denied access to lock a resource, the agent can choose either to continue searching for another resource or block and wait for this resource to become available. In case the agent decides to block on this resource, it informs the host environment of its status. As an agent gets blocked, it can not move or manipulate any other resources neither by acquiring nor releasing them and it can not unblock from waiting for this particular resource. Only the host environment can approve the unblocking of a blocked agent. This characterizes the deadlock definition used in this thesis as a single-request model as discussed in previous chapter.

In addition to the common assumptions mentioned in chapter 3, we assume that the detection agents will remain safe from tampering and will not be corrupted or

[1] It should be mentioned that if it were possible to create more than one agent at the same time (maybe due to parallel processing power etc.) an extra property like the unique process ID may need to be used in addition.

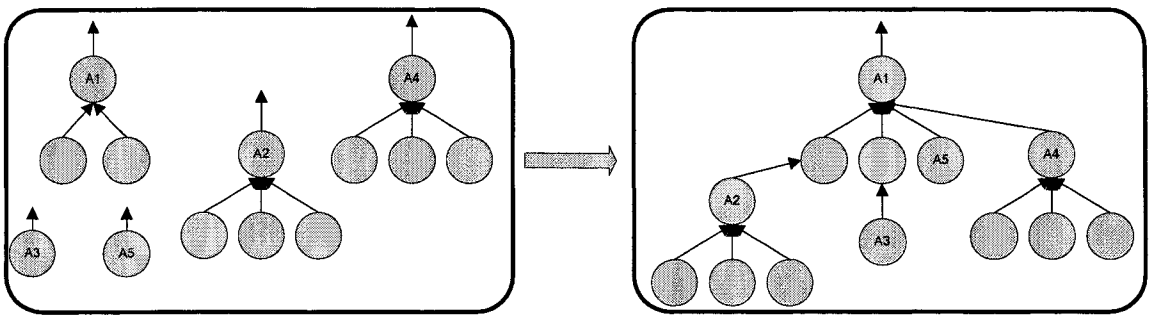


Figure 5.1: Trees and agents in a forest join to form new trees

lying and will reach their destinations within a bounded time.

### 5.1.1 Resource Model

The proposed solution, similar to algorithm SA, utilizes the single resource deadlock model of resource acquisition. This model, as illustrated in section 2.2.2, allows the CA to only request one resource at a time; thus can only create a maximum of one outgoing edge per CA.

In this thesis we refer to wait-for-graph (WFG) to be the set of connected components representing the CAs that are waiting and depending on each other. Due to the single resource property described above, in the absence of deadlocks the WFG can be viewed only as a rooted tree-like graph with one root. At the same time, there might be several rooted WFG trees within the system which are not connected forming a forest. This forest is referred to as the Global WFG.

For each WFG, a CA may join any part of this tree, as illustrated in Figure 5.1.

In addition, other trees in the forest may also join this tree creating new subtrees. This however can only be done via the root of these trees deciding to block on any CA in the main WFG. The root of any WFG remains unaffected until it decides to block on another resource and this decision is what can cause a deadlock.

In the presence of a deadlock, the WFG containing the deadlock cycle will no longer have a root CA available and as a result it can only become the main WFG where other non-deadlocked trees attempt to join it as subtrees.

## 5.2 Algorithm Overview

The proposed solution attempts to detect and resolve deadlocks putting into consideration the extra properties created by the nature of the mobile agents. This algorithm puts higher emphasis on being fully distributed without the need for the existence of a central authority to detect, organize, or resolve deadlocks. The proposed solution is based on the edge-chasing distributed deadlock detection algorithms and concepts described in section 2.3. Edge-chasing algorithms rely on launching "probes" which detect the existence of a deadlock if they returned to their originating node. We ,however, utilize the intelligence introduced in the "probe" agents to provide a more dynamic method for detecting the existence of a deadlock. A deadlock is detected by the "probe" agent when it visits a node twice; as opposed to the traditional method where the probe's return to its originating node is what triggers the existence of a deadlock. The added intelligence to the probe agents also allows them not only to independently decide on the existence of a deadlock but to also take an active role in

resolving it.

### 5.2.1 Solution Entities

The proposed solution requires the existence of three types of mobile agents in the system: consumer agents, detector agents, and information agents.

- **Consumer Agents (CA):** These are the agents assigned to perform a certain task. In doing so, they move in the network locking the resources they need to perform their given task and may get blocked waiting for a resource in the process.
- **Detector Agents (DA):** These agents are created by CAs when they are blocked on a resource. These are light-weight agents that seek CAs who own exclusive lock on a needed resource. In doing so, they keep a list of CAs visited and the resource they are waiting for. The DAs are also responsible for detecting a deadlock if they visit a CA twice in their trip and informing the CAs accordingly. In addition, the DAs will also provide each CA they visit with their own list of known dependant agents and resources in order for that CA to compile a list of 'resources to avoid' in case it becomes the WFG root and start seeking more resources.
- **Information Agents (IA):** These agents have various roles within the deadlock detection and resolution. They are in essence message-like agents that are mainly responsible for communicating information between CAs, DAs, and host environments. A blocked CA creates IAs which are sent to its locked resources

| Entity | Roles and responsibilities |
|---|---|
| Consumer Agent (CA) | • Responsible for performing assigned tasks. It moves through the network seeking and locking the resources it needs to perform this task<br>• Capable of exclusively locking the resources it needs to perform the task<br>• Spawns detector agents (DAs) to initiate deadlock detection<br>• Spawns Information Agents (IAs) to update its location information as soon as it is blocked<br>• Does not otherwise have an active role in the deadlock detection |
| Detector Agent (DA) | • Responsible for deadlock detection by seeking CAs that hold exclusive locks on needed resources<br>• Responsible for deadlock resolution by deciding on the Victim and informing host environments and CAs<br>• Light-weight small search agent<br>• Capable of cloning and destroying itself<br>• Keeps a list of waiting resources and CAs it visited in the process and provides it to CAs it visits as a partial "avoid" list<br>• Responsible for initiating deadlock resolution |
| Information Agent (IA) | • Spawned by CAs and DAs<br>• Extremely light-weight, message-like agent<br>• Reports information as well as its trip time<br>• No active role in deadlock detection |
| Host Environment | • Responsible for hosting mobile agents<br>• Controls and coordinates resource locking and unlocking<br>• The only authority capable of blocking and freeing agents<br>• Takes active role in deadlock resolution by allowing CAs to unblock or free the needed resource |

Table 5.1: Solution entities

to inform the corresponding host environments of its location. In addition, IAs are also used during deadlock resolution to send the unlock request to the

host environment of the resource to be freed and carry back the results. Finally, they can be used for fault tolerance purposes to carry acknowledgment messages between agents.

To address agent mobility problem, CAs inform their locked resources' hosts of their location only when they get blocked waiting for a new resource somewhere in the network. This will allow, DAs to locate the CAs owning exclusive lock on a needed resource and move forward to ancestor CAs.

## 5.3 Deadlock Detection Initiation

Consumer agents (CAs) are assigned to perform certain tasks and in order to do so, they roam the network searching for and seeking to obtain the resources they need. When a CA finds a resource it needs, it may request an exclusive lock on this resource from its host environment. The host environment may either grant or deny the CA's request.

If the CA is denied request for an exclusive lock of a resource, it can either move on searching for alternate resources or block. In this second case, the CA informs the host environment of its decision and blocks. When blocked, CA can neither withdraw its request and unblock nor it can move in the network until either it is granted the resource by the host environment or it is being unblocked by the host environment as part of a deadlock resolution scheme. In addition, the CA can no longer lock or unlock resources while being blocked.

As soon as it is blocked, the CA sends Information Agents (IAs) to notify of its location the host environments of the resources which it owns exclusive locks. This allows the hosts of these resources to inform and direct Detection Agents (DAs) in their search. This location information will only updated when the CA gets blocked again waiting for another resource. The CA neither clears this information when it gets unblocked nor it updates the information as it moves in the network. However, DAs may inform the host environments to clear any information they detect to be outdated. The role of the DAs will be discussed in further detail later in the chapter.

## 5.4 Deadlock Detection

The deadlock detection starts as a CA is blocked for an unacceptable amount of time. This time can be determined initially at the time of CA's creation based on its creator's determination of how much wait time is considered unacceptable. When the CA's waiting time being blocked for a resource exceeds this given time, the CA initiates its own deadlock detection by initiating a Detection Agent (DA) to check if this CA is waiting in a deadlock situation.

The DAs are small light-weight agents that only carry location information of places and resources they visit and can perform minimum tasks. DAs also determine if there is a deadlock. However, since this determination is simply based on visiting the same place twice and not on the need to analyze the WFG, no significant processing or complexity is needed.

As the DA proceeds with its task, the DA carries the following deadlock detection

information:

- Agent ID: The identifier of the CA that this DA is visiting.
- Location: The location where this CA was found blocked. This is also the location of the resource it is blocked on.
- Blocked On: The identifier of the resource where this CA is blocked on.

In addition, the DA queries the blocked CAs as it visits them for the following information to be used for the primitive deadlock avoidance list:

- Primary locks: This is a list of the resources the CA owns exclusive locks on.

Providing this information to the ancestor CAs helps avoiding deadlocks as these CAs now know that attempting to gain exclusive locks on any of these resources would cause a deadlock to form. This deadlock avoidance information will remain valid though incomplete as long as the CA holds the resources it currently owns. This is due to the fact that all the preceding CAs can neither unblock nor release any of their resources. So, as a result, the dependency will remain. There is the possibility however that new agents may arrive at anytime and block waiting for any of the preceding resources along with its dependant resources and blocked agents. These newly arrived CAs will not be known to the DAs that has already moved forward in their search and will not be included in the "*Avoid*" list. For that reason, the deadlock avoidance information can only be considered partial though accurate. Figure 5.2 illustrates this condition where the passing DA did not know of the dependency of

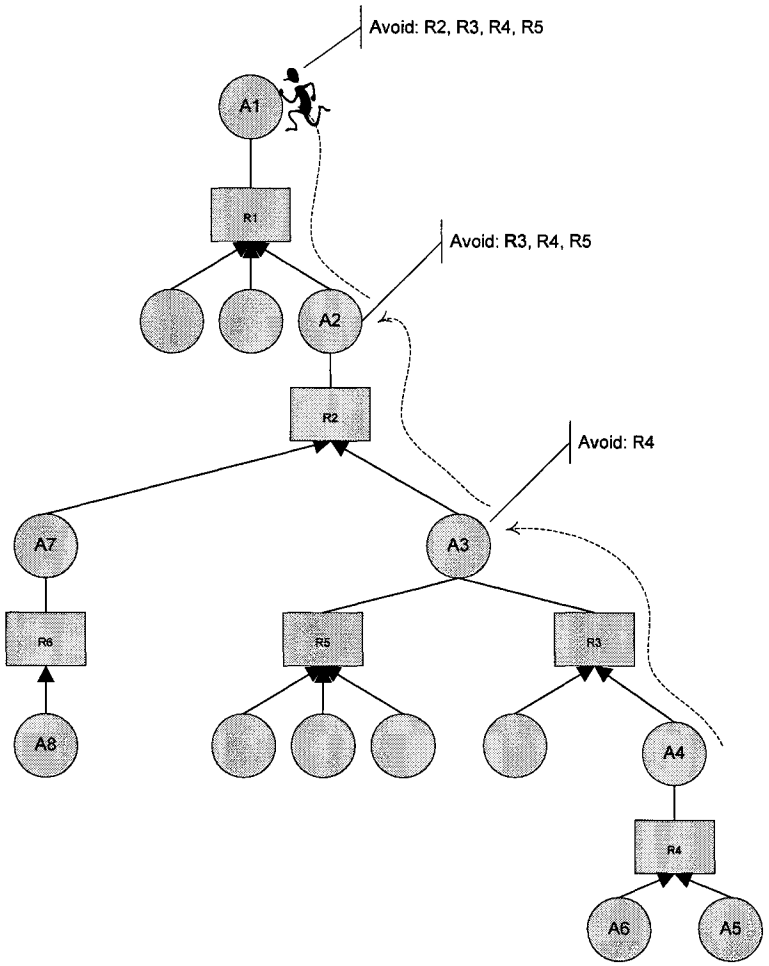


Figure 5.2: DA provides a partial list of resources to avoid

$R_6$ and did not add it to its list. The knowledge of $R_6$ however is expected to be updated as new DAs are spawned (by $A_7$ and/or $A_8$ for example) and passed by $R_6$ to their root.

When initialized, the DA queries the host environment for information regarding the owner of the resource and its location if available. The DA moves to this location and attempts to contact the CA. If the CA was found, the DA provides it with the

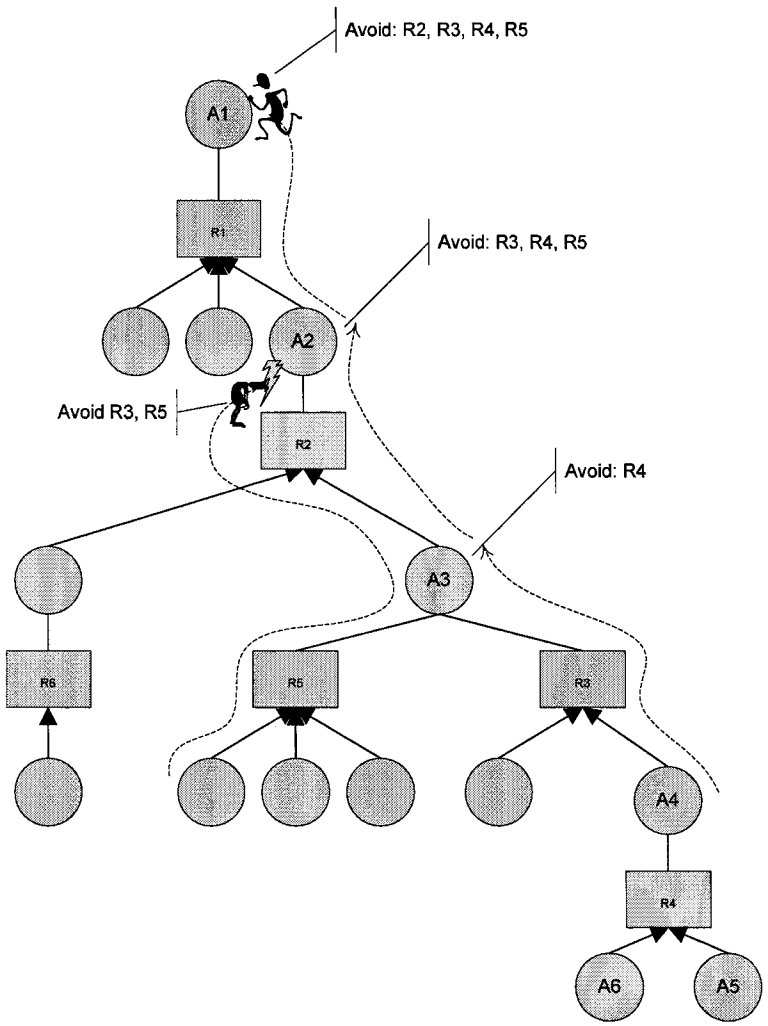


Figure 5.3: DA with known knowledge gets stopped

information it has and checks if any of it is new to the CA.

If the information provided by the DA is all known to the CA visited, then it means that other DAs have passed with similar or more knowledge. In this case, the DA considers its task completed and dies as illustrated in Figure 5.3. Otherwise, if the CA confirms that the given DA does provide new information, then the DA queries it for which resource it is blocked on and which resources it holds exclusive

locks on. The DA then queries the host environment of the location of the CA that owns the lock on this resource and the cycle continues as the DA moves closer to the root of the WFG.

If the DA was not able to locate the CA owning exclusive lock on a resource, the DA will conclude that this is due to one of two possible cases:

1. There is no deadlock and the given CA is the current root of the WFG and is roaming free in the network acquiring resources or processing. Or,
2. This CA is blocked on a resource but the notification IAs has not yet arrived to the host environment.

In both cases, the DA moves back to the host of the last visited resource and waits for either the arrival of an IA updating the blocking information of the CA, or, for the CA to complete its task and free the resource. While waiting, the DA will merge its information list with lists of the newly arriving DAs seeking the owner of the given resource. After merging information, the newly arrived DAs will be disposed of.

As the DA moves upwards in the graph keeping track of agents and resources visited and updating its "avoid" list, visiting an agent for a second time or visiting an agent that is blocked waiting for a resource already in the DA's avoid list will indicate a cycle. The existence of a cycle can only mean that a deadlock exists with the participating CAs and their locations are the ones visited by the DA since it has visited this CA the previous time. These participating CAs can be directly extracted from the DA's itinerary.

In all cases, the initiating CA will not need to send more than one DA every time it is blocked on a resource. This is due to the fact that after sending a DA, all the ancestor CAs will become aware of its dependency and the resources it owns. Hence, if a deadlock existed it will be taken care of by either this given DA or another DA that took over. In addition, if a CA tried to send another DA while still being blocked on the same resource, this newly sent DA will be stopped because it will not have any new information to offer.

## 5.5 Deadlock Resolution

As a DA detects a deadlock due to it visiting a CA twice, the third and final phase of this algorithm is initiated. The deadlock resolution phase deals with the fact that there exists a cycle which must be broken in order to allow the agents to continue with their tasks. At this phase, the DAs are responsible for deciding which resource should be released, inform the corresponding host environment as well as informing all the CAs involved in the deadlock.

Our proposed solution utilizes simple modifications to the algorithms of leader election in a ring to flag CAs involved in a deadlock prior to resolving the deadlock. The deadlock resolution phase can be easily updated to utilize a more enhanced ring based leader election algorithm in order to improve solution complexity in the deadlock resolution phase. To simplify the description of this algorithm, we are using an election process similar to the Chang & Roberts [9] algorithm. Although Chang & Roberts [9] algorithm offers an agent movement complexity of $O(n^2)$ in its worst

case, the average case complexity is $0.69n \log n$

As the deadlock resolution phase starts, the DA initially chooses the CA it visited for the second time as the victim that should release the deadlock-causing-resource it owns and resolve the deadlock. This given resource is the same resource that the previously visited CA is waiting for. This choice of the victim node is based on the fact that there are no pre-assigned priorities for CAs.

The DA starts the deadlock resolution by first creating an artificial "priority" or weight for itself. This is based on the DA's own unique identification, and is intended to allow for the comparison and elimination of other DAs attempting to resolve the same deadlock simultaneously. Since DA's identifications provide a unique and comparable set of values, they provide the needed differentiation factor. This is not an actual priority for the DA but, since the deadlock must be resolved by only one DA, this provides the means to ensure there only exists one DA trying to resolve this deadlock.

The DA starts by informing the CA of the fact that a deadlock was detected and provides the CA with its identifier and priority. From this point forward, the CA will not allow to pass any other DA attempting to break the deadlock which has a lower priority value and that DA will be instructed to terminate. The CAs will also not allow any further DAs attempting to detect a deadlock to pass as long as the deadlock resolution phase is in progress.

Next, since the DA is now able to retrace its steps backwards within the deadlock cycle, it starts doing so while informing CAs along its route of the deadlock resolution

phase and of its priority. If the DA was instructed by any of the CAs along the route that there exists another DA with a higher priority, then this DA will consider its job done and terminate. Otherwise, if the DA makes a full reverse-cycle back to its victim CA it will only mean that the deadlock cycle is now free of other DAs attempting to break the same cycle as well as from any new DAs trying to detect the deadlock and which may erroneously do so if they were allowed at this point.

At this point, it is now safe to resolve the deadlock. So, the DA informs the host environment that has the victim CA blocked to release the resource from the victim CA. The host environment as previously defined is the ultimate authority for unlocking resources. An IA that contains the victim CA's identification and resource information is sent to the corresponding host environment that owns the given resource-to-be-freed requesting the release of the resource due to deadlock resolution. The IA then returns to the initiating host environment with the result of the request. If the release of the resource was successful, then the host environment informs the CA of the removal of this resource from it and informs the DA of the success of the deadlock resolution.

The deadlock is now resolved. However, before considering the job done, there are still two unfinished tasks to be completed by the DA. The first is to update the "avoid" list of the previously-deadlock-involved CAs. Then, the second task is to clear the deadlock resolution status from those CAs in order to allow future and further DAs to detect and resolve any future deadlocks. We rely here on the fact that after the deadlock resolution, all the CAs that were part of the deadlock will become ancestor CAs to the victim CA in the WFG. As a result, in order to achieve those

final tasks, the DA starts moving forward again in a fashion similar to the deadlock detection phase. The DA informs the CAs to clear the deadlock resolution status and to no longer avoid the resource that was freed. The DA continues moving forward until one of the following cases occur:

1. It reaches the agent that took ownership of the new resource. This will mean that all the deadlock-resolution flags have been cleared. If this CA is now blocked on another resource, then the DA will continue from this point forward seeking the WFG root as illustrated in the deadlock detection phase (section 5.4).

2. It reaches the WFG root. This will mean that the further ancestor CAs previously involved in the deadlock have received the resources they needed and finished their tasks. The DA's job is now done as any further deadlock-resolution statuses that were not cleared by the DA are now cleared by the CAs themselves. The DA now switches to deadlock-detection phase and merges with newly incoming DAs as described in the deadlock detection phase.

3. It reaches the point where the CA can not be located. This could be due to any of the conditions described in the deadlock detection phase. From the DA's prospective, this is similar to the previous case since at this point this is the known root meaning that any further ancestor CA has had its information updated.

Once this last step is concluded, the DA will remain at the new WFG root merging its information with any newly arriving DAs and waiting for the possibility of new

ancestor CAs to be created as a result of the root blocking on a new resource or newly arrived information. There is always a need to maintain a DA living at the root to ensure the continuity of this algorithm as new deadlocks may form without the need to have descendant CAs retransmit new DAs periodically to check for the possible new deadlock situation.

## 5.6 Formal Description

In the following we will provide a pseudo code for the proposed solution as described in previous sections. The algorithm is request based hence for every host, CA, DA, and IA, there exists a message handling algorithm in order to respond to requests. Requests are sent to the entity along with the needed parameters utilizing the "request" call.

### 5.6.1 Variables and Overall Structure

The following are some key variables used with the pseudo code:

***ResAvoidV*** Is a vector that carries the list of resources to be avoided in order to avoid creating a deadlock. This list is carried by the CAs and updated and exchanged with every visiting DA. The resources in this vector are also compared to the list of avoid resources carried by visiting DA to determine if the DA has new information and should be allowed to move forward.

***MyResV*** Vector containing the resources locked by a given CA

***DLResFlag*** A flag that is used to indicate if a deadlock resolution phase is in progress. Once this flag is set, the CA will not allow any DAs to proceed unless the DA is attempting to initiate a deadlock resolution phase and it has a smaller id than the current DA which has previously set this flag.

***MyResAvoidV*** Vector of dependant resources compiled by the DA as is traverses the waiting CAs and exchanges information with them.

***MyVisitV*** Vector carried by the DA that contains the list of CAs visited, their location, resource they are blocked on, and the resources the CA owns.

***alive*** A boolean flag that determines if this DA should terminate.

***rcpt*** A boolean flag used by IAs that is used to determine if the IA should wait for a response after it delivers its message.

### 5.6.2 Algorithms

In the following sections we will present a set of pseudo code algorithms, these algorithms are:

***Host Request Handling*** algorithm describes how the host environment handles agents requests including requests to lock/unlock resources and requests for CA to block waiting for a given resource.

***Consumer Agent Main Functionality*** algorithm provides the main execution loop for the consumer agent. This includes moving to and acquiring new resources, blocking when needed to, performing task, then releasing the resources.

***Consumer Agent Request Handling*** algorithm describes how the consumer agent handles requests that are being sent to it by detector agents or host environment. This includes requests for the avoid list, updating/providing information, setting/clearing deadlock resolution flag,...etc.

***Detector Agent Main Functionality*** algorithm provides the main execution loop for the detector agent. This includes moving towards the root of the WFG, deciding if there is a deadlock, and merging with other DAs if found

***Detector Agent Deadlock Resolution*** algorithm describes the functionality and steps taken by the DA to resolve a deadlock cycle. This includes tagging the CAs for deadlock resolution phase and starting a leader election to ensure only one DA attempts to resolve the deadlock. Then, finally, clearing the flag and letting the descendent CAs know not to avoid the newly freed resource.

***Detector Agent Merge*** is the logic that allows detector agents to merge their knowledge lists and decide for themselves if they should continue enroute or die.

***Detector Agent Request Handling*** algorithm allows the DA to respond to requests sent by other DAs to merge their lists together.

***Information Agent Main Functionality*** algorithm is the main execution pseudo code for Information Agents. IAs move to the given destination, delivers the

message, wait for response if they were instructed to do so, and return that response back.

### 5.6.3 Host Environment

**Algorithm 1**: Host Requests Handling

**input** : operation request *oreq*, resource id *resid*, agent identification *aid*, [location *info*]
**output**: result *result*

```
result ⟵ denied
switch oreq do
    case lock
        if Resource(resid).status ≠ locked then
            Lock(resid, aid)
            result ⟵ ok
        break
    case block
        Block(aid)
        WQ[resid].enq ⟵ aid
        result ⟵ ok
        break
    case unlock
        if Resource(resid).status = locked &
        Resource(resid).owner.id = aid then
            Unlock(resid)
            if WQ[resid] ≠ ∅ then
                aid ⟵ WQ[resid].deq
                self.request(lock,resid, aid)
                GrantResource(aid, resid)
                Unblock(aid)
            result ⟵ ok
        break
    case resource-info
        result ⟵ Resource(resid).owner
        break
    case update-info
        if Resource(resid).status = locked ∧ Resource(resid).owner.id = aid
        then
            Resource(resid).owner.location = info
        break
    case resolve
        result ⟵ newIA(aid, info, True, remove-res, resid)
        GetCA(aid).request(remove-res,resid)
        break
    otherwise
        result ⟵ unknown
    Return(result)
```

### 5.6.4 Consumer Agent (CA)

**Algorithm 2**: Consumer Agent (CA) Main Functionality

```
input: vector resources, task task
ResAvoidV ⟵ ∅
MyResV ⟵ ∅
DLResFlag ⟵ False
foreach r ∈ resources do
    if r ∉ ResAvoidV then
        MoveTo(GetHost(r))
        if HostRequest (lock,r,self)⟶ denied then
REM         /*agent may seek alternate resources at this point or:*/
            HostRequest (block,r,self)
            forall elements ∈ MyResV do
                new IA (self, GetHost(elements), False,update-info, self.host)
            WaitFor(timeout ∨ ResourceGranted(self, r))
            if timeout then
                new DA (self)
                WaitFor(ResourceGranted(self, r))
        else
            MyResV ⟵ r
PerformTask (task, MyResV)
foreach r ∈ MyResV do
    MoveTo(GetHost(r))
    HostRequest(unlock, r, self)
```

**Algorithm 3**: Consumer Agent (CA) Request Handling

**input** : operation request *oreq*, [vector *avoidV*], [agent id *aid*],[resource id *resid*]
**output**: result *result*

```
result ⟵ denied
switch oreq do
    case list-update
        if avoidV ⊈ ResAvoidV then
            ResAvoidV ⟵ avoidV ∪ ResAvoidV
            result ⟵ ok
        break
    case list-get
        result ⟵ ResAvoidV
        break
    case res-owned
        result ⟵ MyResV
        break
    case blocked-on
        result ⟵ r
        break
    case set-flag
        if ¬(DLResFlag ∧ (aid < DLResFlag.aid)) then
            DLResFlag ⟵ True
            DLResFlag.aid ⟵ aid
            result ⟵ ok
        break
    case clear-flag
        DLResFlag ⟵ False
        break
    case unavoid-res
        Remove (resid, ResAvoidV)
        break
    case remove-res
        Remove (resid, MyResV)
        Add (resid, resources)
        break
    otherwise
        result ⟵ unknown
    return result
```

### 5.6.5 Detector Agent (DA)

**Algorithm 4**: Detector Agent (DA) Main Functionality

```
input: agent curCA
MyResAvoidV ⟵ ∅
MyVisitV(aid, loc, blockedon, owns) ⟵ ∅
alive ⟵ True
newCA ⟵ ∅
while alive do
    if ((newCA.location ≠ ∅ ∧ curCA.request(list-update, MyResAvoidV) →
    denied) ∨ (curCA.DLResFlag) ∨ (¬Found(curCA))) then
        alive ⟵ False
        break
    MyResAvoidV ⟵ curCA.request(list-get) ∪ MyResAvoidV)
    if Blocked(curCA) then
        blockedOn ⟵ curCA.request(blocked-on)
        MyVisitV.add(curCA, this.host, blockedOn, curCA.request(res-owned))
        newCA ⟵ HostRequest(resource-info, blockedOn)
        if newCA.location → ∅ then
            WaitFor(newCA.location ≠
            ∅ ∨ ResourceGranted(curCA, blockedOn))
        if ResourceGranted(curCA) then
            alive ⟵ False
            break
        MoveTo(newCA.location)
        DAMerge()
        if Found(newCA) then
            curCA ⟵ newCA
            if curCA ∈ MyVisitV then
                DeadlockResolution(curCA)
        else
            MoveTo(curCA)
            DAMerge()
            HostRequest(update-info, blockedOn, newCA.aid, ∅)
            newCA.location ⟵ ∅
    else
REM     /* CA that used to be here is no longer blocked. */
        alive ⟵ False
        break
```

**Algorithm 5**: Detector Agent (DA) Deadlock Resolution

```
input: agent curCA
resolved ⟵ False
targetCA ⟵ curCA
prevEntry ⟵ GetPrev(MyVisitV)
targetRes ⟵ prevEntry.blockedon
repeat
    if CArequest(curCA, set-flag, self.aid) ⇢ denied then
REM     /*Another DA with higher id is doing the job*/
        alive ⟵ False
    else
        MoveTo(prevEntry.loc)
        curCA ⟵ prevEntry.aid
        prevEntry ⟵ GetPrev(MyVisitV)
until (¬alive) ∨ (curCA = targetCA)
if alive then
    HostRequest(resolve, targetRes, targetCA, targetRes.loc)
    for ∀ visited ∈ MyVisitV do
        MoveTo(visited.loc)
        curCA ⟵ visited.aid
        if Found(curCA) then
            CArequest(curCA, unavoid-res, targetRes)
            CArequest(curCA, clear-flag)
        else
REM         /*Job is done, the rest of CAs have moved on.*/
            alive ⟵ False
            break
```

**Algorithm 6**: Detector Agent (DA) Merge

```
    DAV ⟵ HostRequest(get-DAs)
    for ∀DA ∈ DAV do
REM     /*contact other DAs within this host and merge*/
        if DArequest(DA, list-merge, MyResAvoidV, MyVisitV) then
REM         /*DAs belong to the same connected component*/
            if DA.aid > self.aid then
                alive ⟵ False
                break
```

**Algorithm 7**: Detector Agent (DA) Request Handling

```
    input : operation request oreq, [vector avoidV], [vector visitV]
    output: result result

    result ⟵ denied
    switch oreq do
        case list-merge
            if (avoidV ⊆ MyResAvoidV) ∨ (visitV ⊆ MyVisitV)
            then
REM             /*DAs belong to the same connected component, merge*/
                MyResAvoidV ⟵ avoidV ∪ MyResAvoidV
                MyVisitV ⟵ visitV ∪ MyVisitV
                if oreq.owner.aid > self.aid then
                    alive ⟵ False
                result ⟵ ok
            break
        otherwise
            result ⟵ unknown
        return result,[MyResAvoidV],[MyVisitV]
```

### 5.6.6 Information Agent (IA)

**Algorithm 8**: Information Agent (IA) Main Functionality

**input**: agent id *owner*, location *dest*, boolean *rcpt*, operation *oreq*, data *msg*

```
MoveTo(dest)
if rcpt then
    WaitFor (result ⟵ dest.request(oreq, owner, msg))
    MoveTo(GetHost(owner))
else
    dest.request(oreq, owner, msg)
    result ⟵ True
return result
```

## 5.7 Deadlock Detection Example

In this section we illustrate the proposed algorithm using a simple example. In this example there exists four mobile agent host environments $H_1$, $H_2$, $H_3$, $H_4$ all of which have no relation or pre-assigned priorities. There exists 4 CAs in this system: A, B, C, and D roaming in the network and seeking resources to complete their tasks. Resources are denoted $R_1$,$R_2$,...,$R_x$ except in figure illustrations where resources are referred to by the notation $Rx_{(CA)}$ where $x$ is the resource number and $CA$ is the CA holding exclusive lock on this resource. Host $H_1$ contains $R_7$ and $R_8$, $H_2$ contains $R_1$ and $R_2$, $H_3$ contains $R_3$ and $R_4$, while $H_4$ contains $R_5$ and $R_6$.

The following sample sequence of events initiates the deadlock situation:

1. A moves to $H_3$ and locks $R_3$

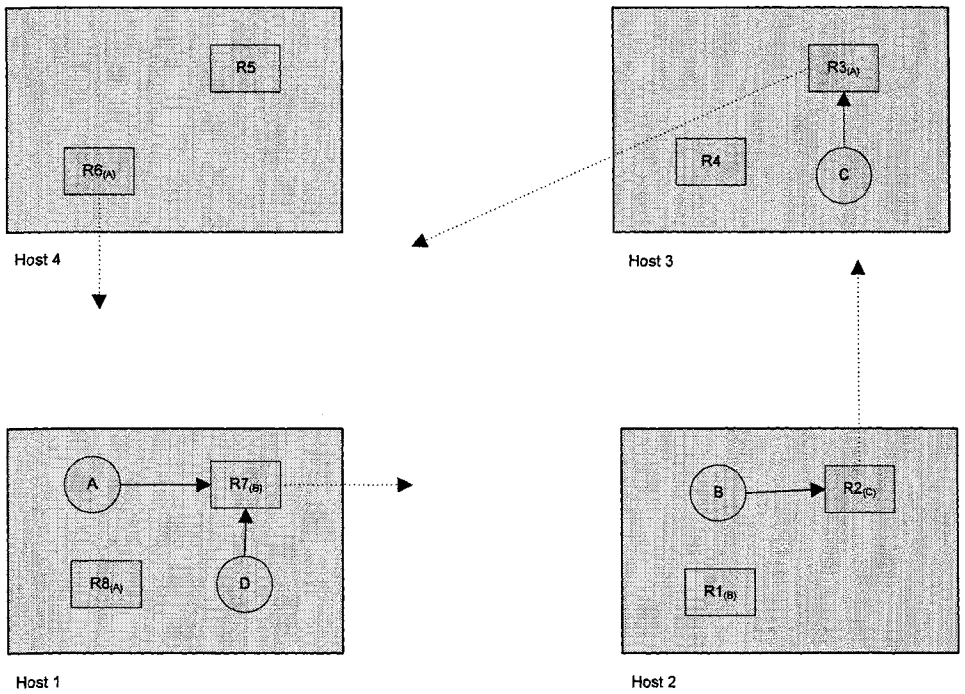


Figure 5.4: Deadlock detection example: deadlock initiation

2. C moves to $H_2$ and locks $R_2$

3. A moves to $H_4$ and locks $R_6$

4. B moves to $H_1$ and locks $R_7$

5. C moves to $H_3$, attempts to lock $R_3$ and gets blocked

6. C sends IA to $H_2$, informing it that it is blocked at $H_3$, the IA will also inform $H_2$ of the trip time it took

7. A moves to $H_1$ and locks $R_8$

8. B moves to $H_2$, attempts to lock $R_2$ and gets blocked

9. B sends an IA to $H_1$, informing it that it is blocked at $H_2$

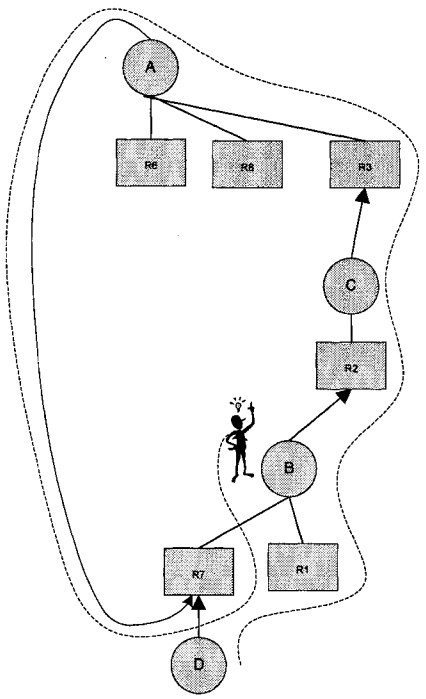


Figure 5.5: deadlock detection logical view

10. A attempts to lock $R_7$ and gets blocked

11. A sends IA to $H_3$, and another IA to $H_4$ informing them that it is blocked at $H_1$

12. D moves to $H_1$, attempts to lock $R_7$ and gets blocked

As a result, the global wait-for-graph looks like the following:

$(A[R_3 , R_6 , R_8], D) \rightarrow B[R_1, R_7] \rightarrow C[R_2] \rightarrow A$

To simplify this example, we will assume that CAs' maximum wait limit set by each agent will expire in the following order: D, B, C, A. As a result, a sample sequence of events will be as follows:

1. D starts by creating a DA and sending it to the owner of $R_7$

2. $DA_D$ moves to $H_1$, locates B, and queries it for the resource it is waiting for and for its primary locks and info

3. $DA_D$ updates its information table as a result:

| Agent | Host | Blocked | Pri. Locks |
|---|---|---|---|
| B | $H_2$ | $R_2$ | $R_1$, $R_7$ |

4. $DA_D$ then queries the host environment $H_2$ for the information of the owner of $R_2$'s primary lock as well as the last know location of it being blocked. The response indicates that C is the owner and the last known blocked location was at host $H_3$

5. $DA_D$ moves to $H_3$ and attempts to contact C

6. At this time, B times out and decides to send a DA of its own. However, realizing that $DA_D$ has already passed by and now carries similar info, $DA_B$ terminates

7. $DA_D$ informs C to avoid $R_1$ and $R_7$ in future resource acquisition as long as it is still holding its current set of locked resources

8. $DA_D$ queries C for its information and updates its table:

| Agent | Host | Blocked | Pri. Locks |
|---|---|---|---|
| B | $H_2$ | $R_2$ | $R_1$, $R_7$ |
| C | $H_3$ | $R_3$ | $R_2$ |

9. $DA_D$ queries the host $H_3$ for owner and location information. The response indicates that A is the owner and $H_1$ is the last known location for it

10. $DA_D$ moves to $H_1$ and attempts to contact A

11. $DA_D$ informs A to avoid $R_1$, $R_2$, and $R_7$ in future resource acquisition as long as it is still holding its current set of locked resources

12. $DA_D$ queries A for its information and updates its table:

| Agent | Host | Blocked | Pri. Locks |
|---|---|---|---|
| B | $H_2$ | $R_2$ | $R_1$, $R_7$ |
| C | $H_3$ | $R_3$ | $R_2$ |
| A | $H_1$ | $R_7$ | $R_3$, $R_6$, $R_8$ |

13. $DA_D$ queries the host $H_1$ for owner and location information. The response indicates that B is the owner and $H_2$ is the last known location for it

14. $DA_D$ moves to $H_2$ and attempts to contact B

$DA_D$ realizes it has visited B before. This can only mean that there is a deadlock situation involving the agents visited since B was last visited, namely C and A as illustrated in figure 5.6. This concludes the deadlock detection phase and initiates the deadlock resolution phase.

Although there is a possibility for a race condition that may occur as a consumer agent gets blocked but has not yet updated its dependant resources while at that time a DA is trying to locate it. This would only happen once as the DA moves back to the resource's location, clears the outdated location information, and waits for location update if and when the CA gets blocked. Figure 5.8 illustrates the condition

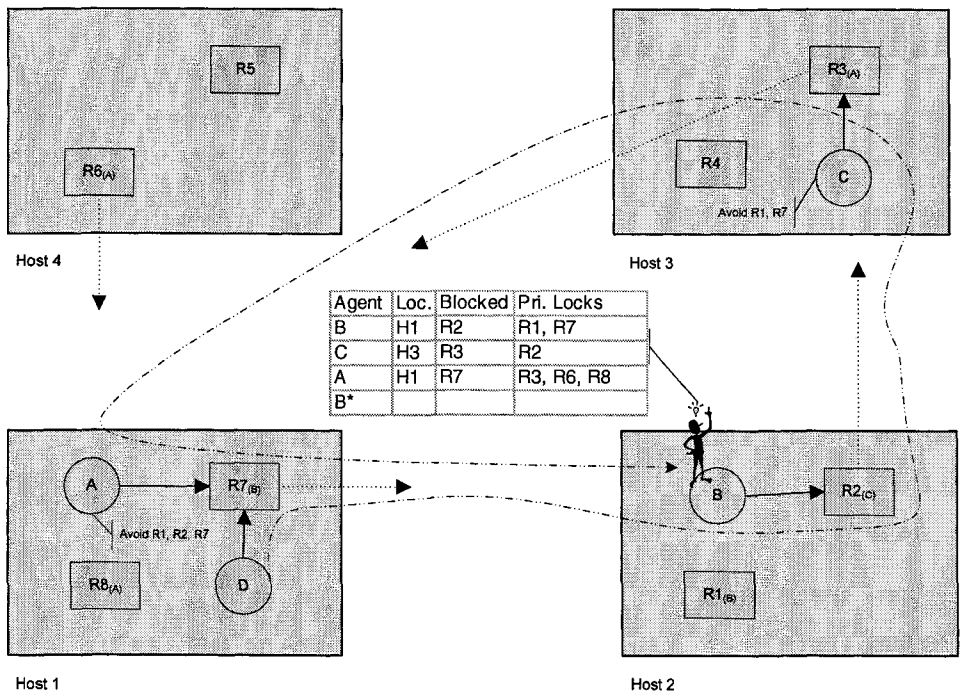


Figure 5.6: deadlock detected when CA is visited twice

described. This situation will only result in one extra round trip migration for a DA between two hosts but will neither affect the efficiency of the algorithm nor will it require it to restart.

Now that the detection phase is concluded, $DA_D$ chooses B as its victim CA since it was the one where the deadlock was detected at. And since the cycle was created due to the dependency of previous agent A on $R_7$, $DA_D$ will attempt to release $R_7$ in order to break this cycle. But first, $DA_D$ must ensure that there are no other DAs attempting to detect or break the deadlock at the same time. $DA_D$ informs B of its priority and that a deadlock resolution is in progress. As a result, B no longer allows any detection agents to move forward attempting to detect a deadlock. In addition, B will no longer allow any detection agent attempting to resolve the deadlock unless it

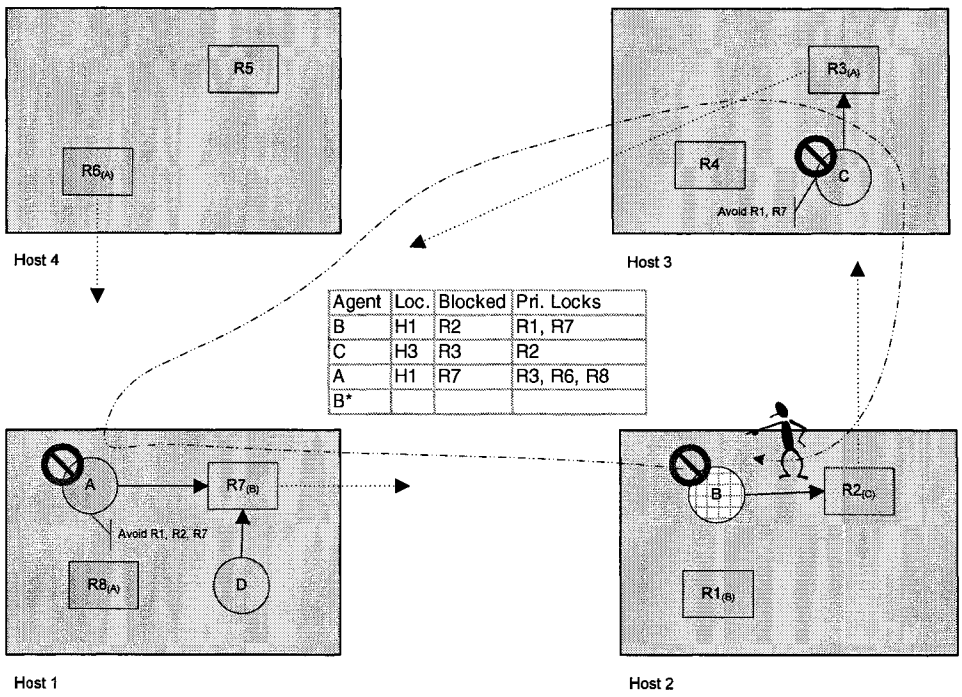


Figure 5.7: Deadlock Resolution: DA chooses victim and clears pending DAs

had a higher identifier than the previous agent. Otherwise, those deadlock resolution DAs will be terminated.

$DA_D$ then starts moving backwards utilizing its resources-and-waiting-agents table it created. This table will stay valid since the CAs are blocked and unable to move. $DA_D$ informs every CA in the list, namely A and C, of the deadlock resolution phase which causes A and C to set their "deadlock resolution" flag. When this flag is set, new detecting DAs will not be allowed to continue into the cycle and any DAs with lower identifier attempting to resolve the deadlock will terminate. Figure 5.7 illustrates the reverse movement of $DA_D$ in order to clear other DAs from the cycle.[2]

[2]As mentioned in previous sections, it is possible for $DA_D$ to be informed by CA in its reverse direction journey of the existence of another DA with higher identifier. In this case $DA_D$ will terminate.

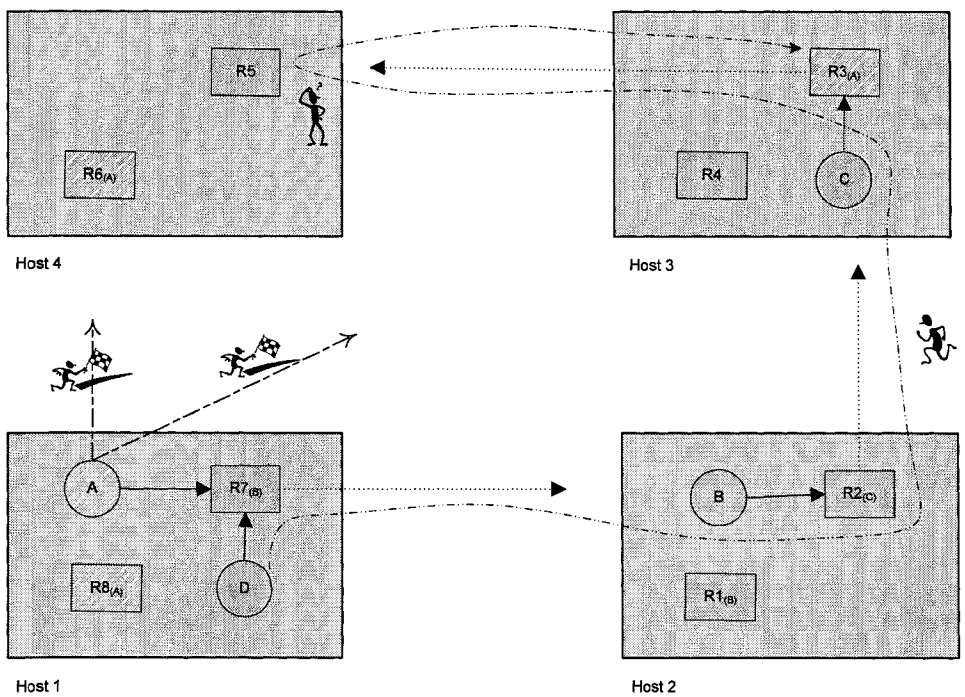


Figure 5.8: Outdated location info situation

If $DA_D$ arrives to its starting point B, it can safely assume it is the only DA within this cycle. As a result, $DA_D$ informs the host environment $H_2$ of the deadlock situation and resolution and provides the ID of victim CA: B, the resource to be unlocked: $R_7$, and its location: $H_1$. The host environment reacts by sending an IA to the corresponding host environment $H_1$ indicating the deadlock resolution situation and requesting the release of B's exclusive lock on $R_7$. If the request succeeded, $H_2$ then informs B that its lock on $R_7$ has been terminated as illustrated by Figure 5.9.

Now $DA_D$ is left with one final cleanup job in order to remove the deadlock-resolution blocks and allow for new DAs to arrive and do their work in the future. In addition, it should also remove the resource that has been freed from the avoid table of CAs. In doing so, $DA_D$ starts moving forwards in a similar fashion to the

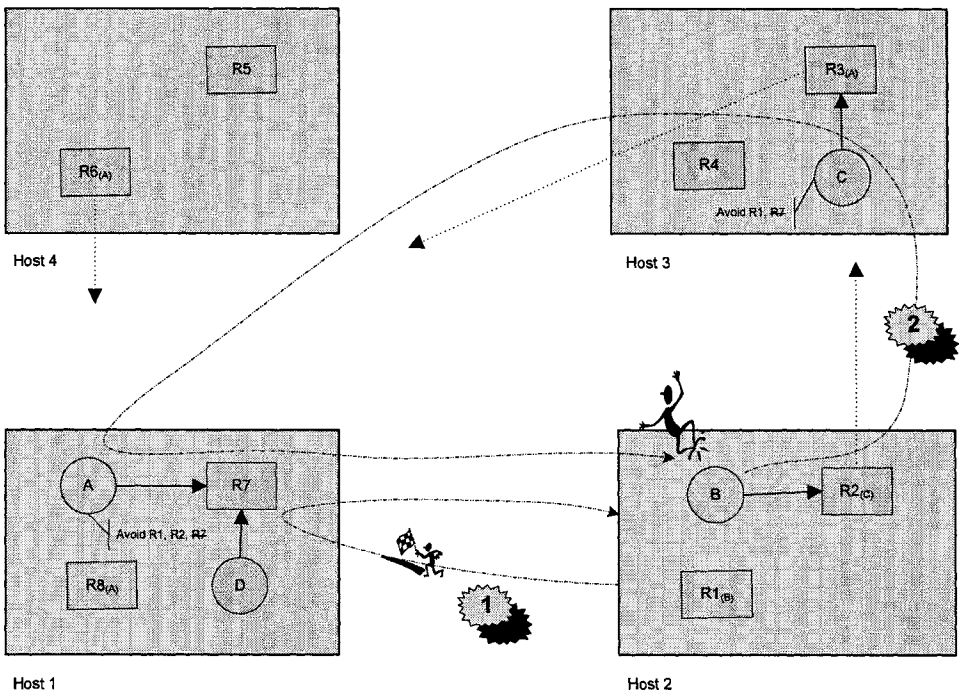


Figure 5.9: Deadlock Resolution: resource freed and flags removed

deadlock detection phase however relying on its own data table that represents the CAs previously involved in the deadlock. The DA attempts to contact every CA in the list and instruct them to clear the deadlock resolution status and to remove the freed resource from their avoid lists. $DA_D$ continues until it informs all the CAs in its list or it fails to contact one of the CAs. In the second case, this will mean that the CA has already received the resource it was blocked upon and was freed. The conclusion of this step indicates the conclusion of the deadlock detection and resolution as in Figure 5.9.

# Chapter 6

# Analysis

## 6.1 Correctness issues

As described in section 2.3, in order to prove the correctness of the proposed algorithm we must show that this algorithm fulfills the progress and safety conditions. In the following subsections we will provide the necessary proof to the correctness of the proposed algorithm.

It should be noted that within this chapter, we will have a special consideration for the possible case of deadlocks forming as a result of two or more trees concurrently join together as illustrated in Figure 6.1. In this special case we will assume that there will always be one last requestor in this sequence *(for example, A1 in Figure 6.1)* resulting in the creation of a new root at some point in time within the deadlock-formation sequence of events.

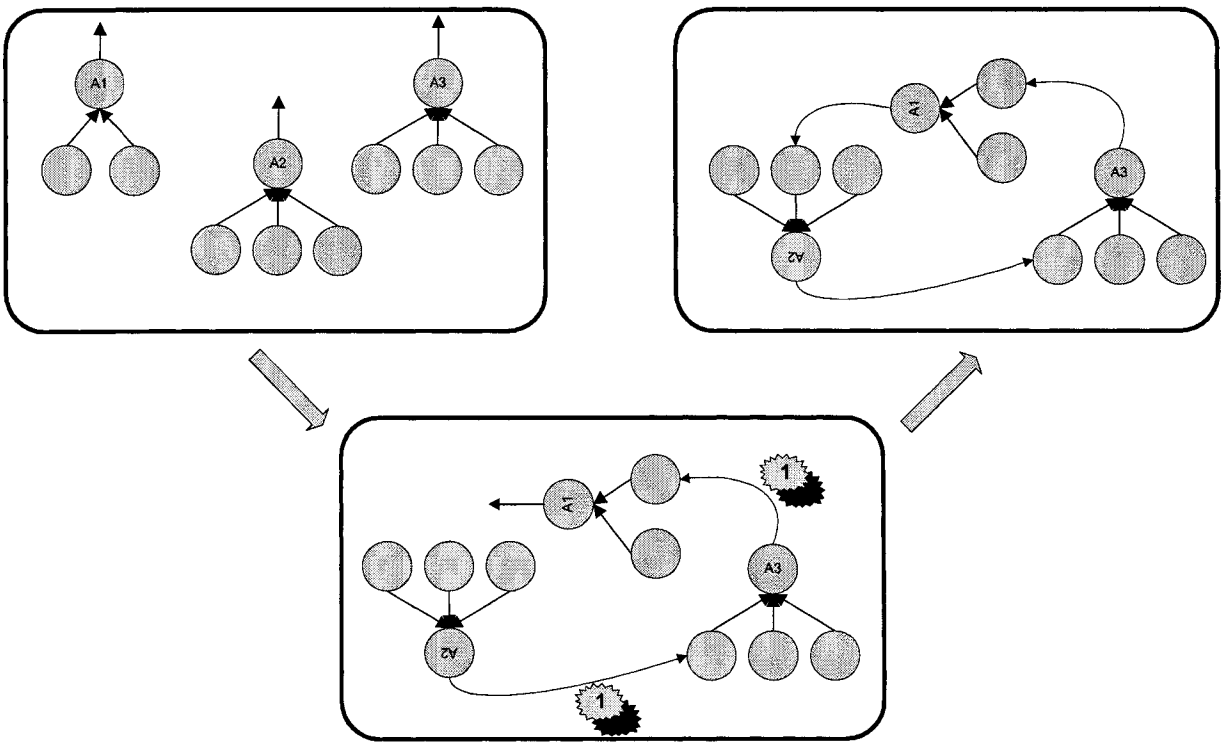


Figure 6.1: Trees could merge to form a deadlock

### 6.1.1 All Deadlocks Detected

In the following we will discuss the progress criteria that is necessary to prove the correctness of the proposed solution. These criteria are concerned with the need to detect any deadlock within the finite time[37].

**Lemma 6.1.1.** *A deadlock cycle can only be created by the consumer agent at the root of the WFG.*

*Proof.* When blocked, a CA can not attempt to acquire any other resource. That situation leaves the only non-blocked CA at the WFG with the ability to attempt to

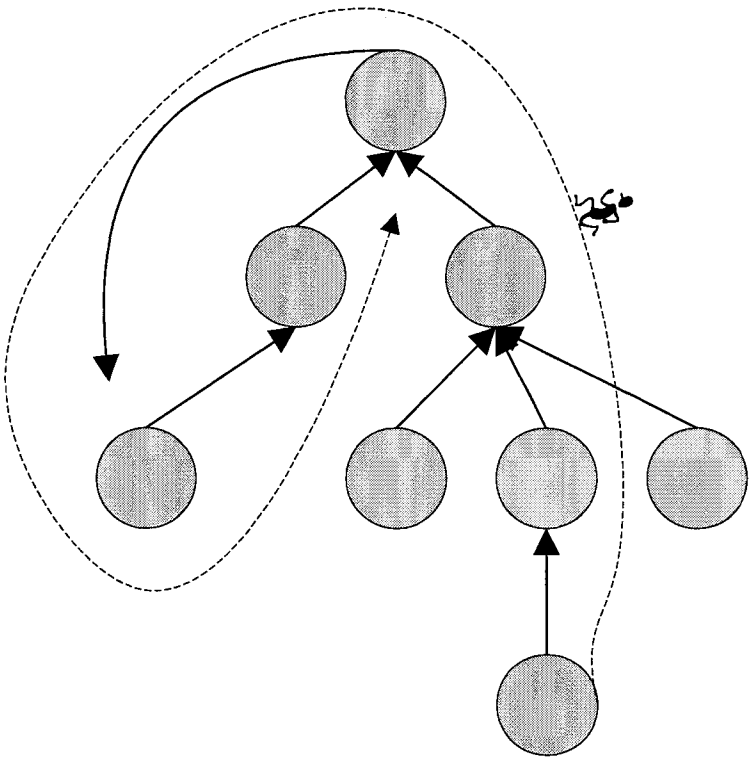

Figure 6.2: Deadlocks can occur only at WFG root

acquire new resources and possibly create a deadlock. This CA can only be the root CA of the WFG. □

The situation is illustrated in Figure 6.2 where a deadlock will be formed if this root consumer agent tried to acquire and block on a resource owned by a dependant blocked agent.

**Lemma 6.1.2.** *At any given point in time, there can exist a maximum of one deadlock in the WFG.*

*Proof.* Since only the root CA can create a deadlock cycle by blocking on a dependant

resource, there will be no more free CAs in the WFG that can attempt to acquire further resources. Hence, once a deadlock cycle is created, no other cycle can appear in the WFG. □

**Lemma 6.1.3.** *If a deadlock exists, at least one DA will detect it*

*Proof.* All DAs seek the root of the WFG despite their starting position. Since deadlock cycles occur only from the root of the WFG, any DA is bound to reach the deadlock cycle given that this DA carries new information no other previously passing DA has. □

**Theorem 6.1.4.** *All deadlocks are detected.*

*Proof.* By Lemmas 6.1.1, 6.1.2, 6.1.3. □

### 6.1.2 No Phantom Deadlocks

In order to prove the correctness of distributed deadlock detection algorithms, it is necessary to satisfy the safety criteria by demonstrating that the algorithm does not detect or attempt to resolve phantom deadlocks.

**Lemma 6.1.5.** *Phantom deadlock can only occur iff two or more DAs attempted to resolve the same deadlock.*

*Proof.* Once a deadlock existed, by Lemma 6.1.2, there can only be a maximum of one at a time and as a result phantom deadlock can not occur as a result of resolving another deadlock. And since blocked CAs can not spontaneously abort or unblock

hence can not remove any of their edges, that leaves only one possibility for an edge to be removed from a deadlocked WFG which is via a deadlock resolution attempt which is the responsibility of the DAs. Hence, unless more than one DA attempted to remove more than one edge of the deadlock cycle there is no other possibility for a phantom deadlock to occur. □

**Theorem 6.1.6.** *No phantom deadlocks are detected*

*Proof.* By Lemma 6.1.5, it is sufficient to prove that no more than one DA can resolve the same deadlock. As described in section 5.5 when any DA detects a deadlock and prior to resolving it, an election is being done that ensures only one DA will be responsible for resolving the cycle. □

### 6.1.3 Deadlock Avoidance

The proposed solution provides a technique for avoiding known dependent resources. It is necessary to prove that post deadlock resolution, the freed resource can be safely cleared from this list.

**Lemma 6.1.7.** *Despite which CA was chosen to release its resource for deadlock resolution, the freed resource will no longer need to be avoided by other agents belonging to the WFG*

*Proof.* For any chosen CA that is forced to free one of its resources, the dependant agent that was blocked waiting for this resource receives it and becomes the new WFG's root CA. And since only the WFG-root agent is the one capable of creating

new outgoing edges and acquiring resources while this given resource is still held by it, none of the previously ancestor agents - now dependant agents in the new WFG - can attempt to acquire this resource until they are unblocked. In this case, it will be after the WFG-root agent has finished its task and the resource was freed. Hence, this resource is no longer part of known dependency in the graph. □

## 6.2 Complexity Analysis

From the observations made in this work regarding the behaviour of the single resource model in combination with the assumptions of 2PL and the no-spontaneous-aborts of blocked agents it can be safely stated that the cost of deadlock detection in the proposed solution varies only if a deadlock exists or not. In the following we will look at the cost of detection in these two cases as well as the cost of resolving a detected deadlock.

### 6.2.1 Detection Cost in Deadlock-Free Situation

In the proposed solution, the deadlock detection phase requires that for any blocked consumer agent, say $C_x$, that initiates deadlock detection, a detector agent $D_x$ will be launched and will seek to reach the root of the WFG. If the root was reached then $D_x$ will conclude that there is no deadlock and will merge with currently existing detector agent(s) in there if any. So, in the worst case, $C_x$ is located at the farthest point from the root of the WFG and will have to reach the root with a number of

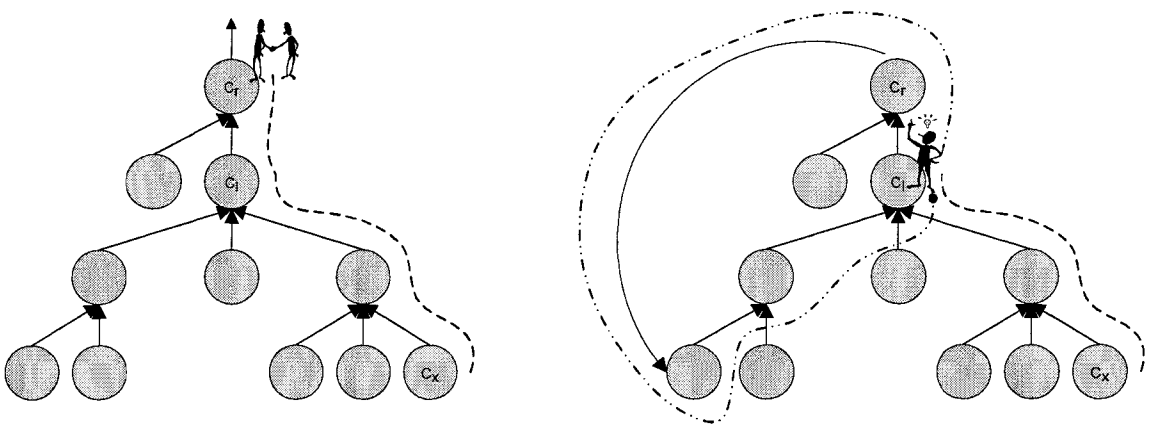


Figure 6.3: Cost of Deadlock Detection

movements $\leq$ the height of the graph.

**Observation 6.2.1.** *Let $C_r$ be the Consumer Agent at the root of the WFG, and let $C_x$ be a leaf CA. The cost for $C_x$ to detect if there exists a deadlock is in the worst case equal the distance $d(C_x, C_r)$ where d is the height of the WFG*

### 6.2.2 Detection Cost When a Deadlock Exists

**Observation 6.2.2.** *Let $C_1 \rightarrow ... \rightarrow C_{i-1} \rightarrow C_i \rightarrow ... \rightarrow C_k$ be a deadlock cycle of k consumer agents. Let $C_x$ be a blocked consumer agent in the WFG which had launched a detector agent $D_x$ and let $C_i$ be the entry point into the cycle for $D_x$.. The cost of deadlock detection is in the worst case equals $d(C_x, C_i) + k$*

In order to understand the validity of this observation, we need to recall the proposed solution. In order to detect a deadlock, a detector agent $D_x$ will need to visit a given consumer agent, say $C_i$, twice. This can be seen as two phases:

1. $D_x$ moves from where it was launched by $C_x$ until it reaches the entry point of the cycle $C_i$ at a cost of distance $d(C_x, C_i)$

2. $D_x$ traverses the cycle until it once again reaches the entry point $C_i$ at a cost of $k$ moves.

At this point the deadlock will be detected and the resolution phase begins.

### 6.2.3 Cost of Deadlock Resolution

For deadlock resolution, we need to recall that only CAs involved in the cycle take part in the election process to choose and resolve the deadlock.

**Observation 6.2.3.** *Let $C_i$ be a Consumer Agent involved in cycle $C_1 \rightarrow ... \rightarrow C_{i-1} \rightarrow C_i \rightarrow ... \rightarrow C_k$. Then the cost of resolving this deadlock is the cost of choosing a leader DA to resolve the deadlock.*

Electing a leader in a ring is a heavily studied field in the distributed computing literature. The proposed solution would require some modifications to the standard algorithms to accommodate the mobile agents' nature and movement. However, complexity would still be of similar cost. The following table summarizes some of these leader election algorithms that could be used:

Therefore, the best solution known so far requires at most $1.271k \log k + O(k)$ movements.

Table 6.1: Leader Election in Asynchronous Rings

| Year | Author(s) | Complexity* | Reference |
|---|---|---|---|
| 1977 | Le Lann | $O(k^2)$ | [38] |
| 1979 | Chang & Roberts | $O(k2)$ | [9] |
| 1980 | Hirschberg & Sinclair | $8k\lceil \log k \rceil + 8k$ | [26] |
| 1981 | Dolev *et al.* | $1.356k \log k$ | [15] |
| 1982 | Peterson | $1.44k \log k + O(k)$ | [45] |
| 1994 | Higham & Przytycka | $1.271k \log k + O(k)$ | [25] |

*: movement complexity, worst case

## 6.3 Comparison With Previous Solution

In the following, we will describe some main differences between proposed solution and the currently existing solution for detecting deadlocked mobile agents.

### 6.3.1 Solution Cost Comparison

Table 6.3.1 describes the main differences between algorithm SA for mobile agents' deadlock detection and the proposed solution. In addition to improved resource utilization due to the elimination of the need for shadow agents as mentioned in a previous section, the proposed solution generally requires less number of detection agents to initiate and detect deadlocks. This makes it more efficient in terms of resources used.

Algorithm SA's number of detection cycles varies by the number of resources held by the consumer agent in addition to the race condition that may occur when consumer agent is holding 3 or more resources. This condition has been illustrated in previous chapter in more details. The proposed solution however requires only one

detection cycle despite the number of resources held. Mainly because this solution follows the owners of the resources held which is always a one-to-one relation.

| | **Current Solution** | **New Solution** |
|---|---|---|
| **Number of consumer agents** | 2 (CA and Shadow agents) | 1 |
| **Number of DAs per CA** | $= r$ | 1 |
| **Total number of detection agents (DAs)** | $n * r$ | $\leqslant n$ |
| **Number of detection cycles for detecting a single deadlock** | $\geqslant r$ | 1 |
| **Number of DA moves per CA (detection phase) (Best)** | $2 * r$ | 1 |
| **Number of DA moves per CA (detection phase) (Worst)** | $\infty$ | $d$ |

Table 6.2: Summary of Comparison Between Current and Proposed Solutions

$n$ number of CAs in the WFG

$r$ number of resources owned by a CA

$R$ total number of resources in the WFG $(n * r)$

$d$ depth of WFG

$C$ number of repetitions to detect a single deadlock

$x$ number of times root WFG is blocked for unacceptable time

A detection agent in algorithm SA needs to move to the resource held by the consumer agent to detect and exchange information with detection agents initiated by other consumer agents who are waiting for that particular resource. Afterwards, that detection agent returns to its initiating shadow agent to report its findings. The same process is required for every resource held by the consumer agent and is usually done by cloning detection agents such that there is a detection agent per resource. In addition, detection agents may be required to repeat the detection cycle if no conclusive results were reached [2]. As a result, in the best case, the detection agents will need to do 2 moves per agent (move to resource, move back) creating a total number of trips equal to double the number of resources. In the worst case, all shadow agents involved will simultaneously initiate deadlock detection cycles and each would require more than one detection cycle per agent. This would total to: (2 * total number of resource held by total agents involved * number of detection cycles). In addition, if there is no deadlock, the detection cycles could repeat infinitely.

The proposed solution on the other hand requires only one detection agent that will move upwards to detect if there is a deadlock by following the edge-chasing mechanism. If the detection agent failed to detect a deadlock from tracing the owners of the resource it is waiting for, then these results are conclusive and the agent would wait at the root point of the WFG (which is also the only point where a deadlock can be formed) and no further future detection will be needed. If other consumer agents simultaneously decided to initiate a deadlock detection cycle, they will be blocked. As a result, there is a best case of 1 move required to reach a conclusive result whether a deadlock exists that involves the given consumer agent or not.

### 6.3.2 Transaction Priorities

The current solution relies on pre-assigned globally unique and comparable priority for each consumer agent. This priority value is needed in order for the solution to be able to execute. Algorithm SA does not go into detailing how to create and maintain this unique priority across all consumer agents in the network. This dependency would require global coordination mechanisms to assign and compare priority for every consumer agent which does reduce algorithm SA's real-life applicability.

The proposed solution is a priority-free solution that does not require to have priority assignment requirements in order to function. This makes the proposed solution more realistic for applications in large networks with many roaming consumer agents. In addition, it frees the system from the overhead and processing needed to arrange and coordinate the assignment of globally unique priorities for every consumer agent which is a burden not currently accounted for in algorithm SA.

### 6.3.3 Methodology and Detector Agents

Algorithm SA utilizes the path pushing scheme for detecting deadlocks for mobile agents. By doing so, each consumer mobile agent, technically its shadow, launches a detector agent for every resource owned in order to initiate the deadlock detection cycle. Hence the total needed number of detector agents = $\sum_{Blocked\ CAs}^{Resource\ owned}$

The proposed solution builds on the edge chasing principles of deadlock detection.

By doing so, each consumer agent launches only one detector agent in order to initiate the deadlock detection. In addition, further elimination process takes place for detector agents to minimize the number of roaming DAs. Hence, the number of DAs used by the proposed solution $\leq$ *Blocked CAs*.

### 6.3.4 Deadlock Detection Initiation

The deadlock detection phase is initiated by algorithm SA as soon as a consumer agent is blocked waiting for a resource. In that case, the corresponding shadow agent immediately launches detector agents for every owned resource to check for deadlock detection information. The design of the algorithm does require that all shadow agents involved in a deadlock to launch detector agents in order for this deadlock to be detected. This is due to the dependency of each shadow agent and its detector agents on the exchange of deadlock detection information with the other agents that are blocked waiting or the resource they own.

The proposed solution introduces a timeout mechanism to optimize detector agents launching. Since the proposed solution provides autonomy for detector agents to detect and resolve deadlocks, there is no dependency on other agents or other detector agents. As a result, by having an arbitrary timeout value, it will help avoiding that all or many consumer agents will launch detector agents simultaneously which further optimizes the number of used detector agents in the system.

### 6.3.5 Resource, Computational & Cost Footprint

As described in a previous chapter, algorithm SA for deadlock detection in mobile agents requires a clone of the main consumer/worker agent, namely "Shadow" agent, in order to handle the processing required by the deadlock detection cycles as they repeat infinitely unless a deadlock is detected. The Shadow agent is also responsible for building the global WFG by analyzing and concatenating the information gathered by detector agents in addition to analyzing the global WFG to find cycles that denote deadlocks.

The proposed solution however does not require the use of a shadow agent since the required processing for deadlock detection is rather light and can be directly performed by the lightweight detection agents. This approach greatly improves resources and processing usage requirements for deadlock detection.

In addition, with algorithm SA once the deadlock detection is initiated, there is no way to know if there were no deadlock. As a result, the deadlock detection phase will continue to execute nonstop until a deadlock is formed and detected or the resource is granted. This processing and repeated deadlock detection cycles carried on by all waiting consumer agents can significantly jeopardize the processing capabilities of the hosts involved as well as the network resources due to large amount of detector agent migrations continuously.

On the other hand, proposed solution is able to conclude if there were or not a deadlock in the WFG. It is also capable of knowing where in the WFG a future deadlock can form. As a result, only one deadlock detection cycle is needed and in

the case that there is no deadlock present a DA waits at the point where a future deadlock can form.

To summarize:

1. Proposed solution's resource footprint is less than half the algorithm SA due to the elimination of the shadow agent requirement and the need for less number of detector agents.

2. Proposed solution's computations reduce CPU utilization as it does not require the building of the global WFG by every consumer agent and analyzing it for cycles.

3. Proposed solution also reduces CPU and network utilization due to its capability to detect that there is no deadlock and its ability to 'park' a DA where a deadlock can occur. As a result, there are no repeated deadlock detection cycles.

# Chapter 7

# Conclusions

## 7.1 Summary Of Contributions

In the following we will attempt to summarize the original new work introduced in this thesis. The thesis is mainly concerned with developing a new algorithm for detecting deadlocked mobile agents. In addition, new ideas were developed while working on this new solution.

### 7.1.1 New Algorithm For Detecting Deadlocked Mobile Agents

This thesis has introduced a new mobile agents solution for distributed deadlock detection which applies to detecting deadlocked mobile agents. This solution is based

on the adaptation and enhancement of distributed edge chasing algorithms. In addition, we attempt to remove the need and dependency on a global preassigned priority for consumer agents while taking into account the consumer agents' movement in the network without a pre-set itinerary.

### 7.1.2 Bounded Cost Solution

The proposed solution is of bounded and predictable cost of movement to detect the existence (or nonexistence) of a deadlock. This contribution is of significant importance in comparison to currently existing solution due to the inability of algorithm SA to detect deadlock-free situations which forces it to continuously repeat execution until a deadlock is formed in the WFG if any. Thus the cost of the existing solution is unbounded.

### 7.1.3 New Detection Paradigm

In traditional probe based deadlock detection algorithms that we are aware of, probes do not have the computational power to actually detect or make decisions regarding deadlocks. As more modern algorithms started utilizing mobile agents as detection probes, they continued to use the traditional frame of mind that relies on the originating transactions/nodes or similar to do the processing. As a result, the detection agents' intelligence is not utilized in deadlock detection decision, instead, the originating transactions are the ones that process data and decide if there exists a deadlock and if it should be resolved. Detector agents in these solutions function only as

enhanced *'probes'*. For example, Krivocapic [36] algorithm for utilizing deadlock detection agents sets the responsibility of deciding which DAs to merge as well as the decision on deadlock existence to the transactions as they get the information from DAs.

In algorithm SA for detecting deadlocked mobile agents [2] the usage of detector agents continues to use the same paradigm. Detector agents move to collect information then return to the originating consumer agent with the data collected. Consumer agents are the ones who decide if there is a deadlock and on whom to resolve it.

The proposed solution however utilizes the intelligence capabilities of detector mobile agents and allow them to make the decisions on whether there is a deadlock and to resolve this deadlock. This paradigm relieves the *'probes'* from having to return and report to originating consumer agents.

### 7.1.4 Partial Deadlock Avoidance

The proposed solution introduces a partial deadlock avoidance technique that is resource and computation efficient. The purpose of this technique is not to fully avoid deadlock creation which is an expensive task but instead to reduce the possibility of deadlock occurrence in the WFG without consuming extra computational power or environmental resources. This is achieved by creating an inventory of known dependent resources by DAs as they visit CAs on their deadlock detection cycle. This list eventually reaches the CA at the root of the WFG which allows it to avoid blocking on these dependant resources.

The technique remains partial because we do not attempt to build a global picture of the WFG so it is dependant on the path taken by DAs as they are traveling to the root of the WFG gaining knowledge as they go of the waiting CAs and the resources they own. In addition, it is always possible for new CAs with their own set of dependent resources and CAs to join the WFG at any point adding a new set of resources that should be avoided but not yet known to the root CA.

However, this technique guarantees that after a deadlock resolution, no deadlock will occur again that consists of the same set or subset of the agents that were part of the previous deadlock. In addition, this method would provide full deadlock avoidance if the following conditions occurred:

1. No new dependant CAs were added to the WFG,
2. All CAs in the WFG have sent a DA,
3. All DAs have either reached their destination to the root of the WFG or were stopped on the way, i.e. there are no DAs left enroute.

In this case the CA at the root of the WFG will have a complete list of dependant resources in the WFG that would cause a deadlock if it tried to acquire any of them.

### 7.1.5 DA Optimizations

In this thesis we have introduced DA optimization techniques that are intended to improve on the number of DAs used and the total number of moves in the average case.

### DA Elimination

As described in 5.6 and illustrated in Figure 5.3, DAs exchange their own knowledge of waiting CAs and dependant resources with the visited CAs. A given DA is only allowed to continue in its route only if it has new information that should be passed otherwise the DA is terminated since another DA has already passed carrying the same information or more.

The DA elimination serves as a method for optimizing the number of DAs and DA movements in the detection phase as unneeded DAs are removed. In addition, if a dependant CA sends a DA for deadlock detection which ends-up visiting an ancestor CA that has not yet sent its own DA, this CA will no longer need to send a DA since all the info it knows has been passed to the visiting DA.

### DA Initiation Timeout

When a CA decides to block waiting for a resource, there is a period of time that must elapse first before this CA decides to send a DA to check if it is in a deadlock situation. This timeout varies by CA and can be decided at design time or can be set as a random value. This serves in further randomizing the creation of DAs within a connected component particularly in heavily populated systems where many CAs may block simultaneously within one connected component.

This streamlining of DA creation within the WFG allows for more efficient usage of DAs information in the average case as DAs will have more time distribution. In addition, the variable timeout allows for early stopping of DAs that have similar info

to what has been collectively gathered by previous DAs.

#### Limited Number of DAs

Within any given connected component, any CA is only allowed to either send one DA or none (as a result of DA elimination previously mentioned). Any given CA will never send another DA as long as it is blocked on the same resource. This limits the number or DAs to be in worst case equal to the number of CAs but it is expected to be much less in average case due to the previously described optimizations.

## 7.2 Future Work

There are several research approaches to be followed based on the work completed in this thesis. The work and solution proposed here are mainly concerned with proposing a new bounded-cost solution for detecting deadlocked mobile agents as well as observing the behavior of singe-resource deadlock model.

Providing experimental verification and performance improvement is a possible future study area. A simulation for the performance of algorithm SA in average case and the possibility for further optimization to the number of DAs and the initiation rules is suggested.

Further study into the deadlock avoidance techniques is also suggested as possible future work. Designing techniques to better improve the proposed partial deadlock avoidance while maintaining small and efficient agents and movement complexity can

prove to be a challenging task.

The expansion of the proposed solution to other deadlock models like the AND model and the OR model is also suggested. The proposed solution along with currently existing solution are mainly reliant on the properties and characteristics of the single-resource model. Expanding these solutions and working towards providing an enhanced new solution for other models like the AND model is a suggested field of study

In addition, further study into the fault tolerance techniques of mobile agents particularly in relation to the proposed solution is another suggested future work. The field of fault tolerance and black hole detection is a widely known and studied field in the mobile agents community today and this work can be applied to protecting the consumer and detector agents in the proposed solution.

The proposed solution can also be modified to utilize various leader election techniques which can be studied as ways to improve the complexity of the solution.

# Bibliography

[1] Kaizar Abdul Husain Amin. Resource efficient and scalable routing using intelligent mobile agents. Master's thesis, University of North Texas, 2003.

[2] Bruce Ashfield. Distributed deadlock detection in mobile agent systems. Master's thesis, Carleton University, 2000.

[3] Bruce Ashfield, Dwight Deugo, Franz Oppacher, and Tony White. Distributed deadlock detection in mobile agent systems. In *IEA/AIE '02: Proceedings of the 15th International Conference on Industrial and Engineering, Applications of Artificial Intelligence and Expert Systems*, pages 146–156, London, UK, 2002. Springer-Verlag.

[4] H. Baala, O. Flauzac, J. Gaber, M. Bui, and T. El-Ghazawi. A self-stabilizing distributed algorithm for spanning tree construction in wireless ad hoc networks. *Jornal of Parallel Distributed Computing*, 63(1):97–104, 2003.

[5] Gabriel Bracha and Sam Toueg. A distributed algorithm for generalized deadlock detection. In *PODC '84: Proceedings of the third annual ACM symposium on*

*Principles of distributed computing*, pages 285–301, New York, NY, USA, 1984. ACM Press.

[6] G. Cabri, L. Leonardi, and F. Zambonelli. Mobile agent technology: Current trends and perspectives. In *Proceeding of the AICA Annual Conference*, November 1998.

[7] K. M. Chandy and J. Misra. A distributed algorithm for detecting resource deadlocks in distributed systems. In *Proceedings of the first ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, pages 157–164. ACM Press, 1982.

[8] K. M. Chandy, J. Misra, and L. M. Haas. Distributed deadlock detection. *ACM Transactions on Computer Systems (TOCS)*, pages 144–156, 1983.

[9] Ernest Chang and Rosemary Roberts. An improved algorithm for decentralized extrema-finding in circular configurations of processes. *Communications of the ACM*, 22(5):281–283, 1979.

[10] E. G. Coffman, M. Elphick, and A. Shoshani. System deadlocks. *ACM Computing Survey*, 3(2):67–78, 1971.

[11] George Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed Systems: Concepts and Design.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1994.

[12] Jonathan Dale. *A Mobile Agent Architecture To Support Distributed Resource Information Management.* Phd thesis, University of Southampton, 1997.

[13] S. Das, P. Flocchini, A. Nayak, and N. Santoro. Distributed exploration of an unknown graph. In *Proc. of 12th Int. Colloqium on Structural Information and Communication Complexity (SIROCCO 05)*, May 2005.

[14] Jose Ramon Gonzalez de Mendovil, Federico Farina, Jose Ramon Garitagoitia, Carlos F. Alastruey, and J. M. Bernabeu-Auban. A distributed deadlock resolution algorithm for the and model. *IEEE Trans. Parallel Distrib. Syst.*, 10(5):433–447, 1999.

[15] D. Dolev, M. Klawe, and M. Rodeh. An $o(n \log n)$ unidirectional distributed algorithm for extrema finding in a circle. Technical Report IBM Research Rep. RJ3185, IBM Corp., San Jose, CA, USA, Jul 1981.

[16] Ahmed K. Elmagarmid. A survey of distributed deadlock detection algorithms. *SIGMOD Rec.*, 15(3):37–45, 1986.

[17] W. Farmer, J. Guttman, and V. Swarup. Security for mobile agents: Issues and requirements. In *Proceedings of the National Information Systems Security Conference*, pages 591–597, 1996.

[18] P. Flocchini, E. Kranakis, D. Krizanc, F. Luccio, N. Santoro, and C. Sawchuk. Mobile agents rendezvous when tokens fail. In R. Kralovic and editors O. Sykora, editors, *In Proceedings of Structural Information and Communication Complexity, 11th International Colloquium , SIROCCO 2004*, volume 3104, 2004.

[19] P. Flocchini, E. Kranakis, D. Krizanc, N. Santoro, and C. Sawchuk. Mobile agent rendezvous in a ring. In *Proceedings of 6th of Latin American Theoretical Informatics (LATIN 2004)*, volume 2976, pages 599–608.

[20] Russel Ernest Frew, Kenneth Robert Whitebread, and Henry Himdle Mendenhall. Information exchange by intelligent mobile agents in a network. US Patent US006009456, December 1999.

[21] L. Gasieniec, E. Kranakis, D. Krizanc, and X. Zhang. Optimal memory rendezvous of anonymous mobile agents in a uni-directional ring. In *In proceedings of SOFSEM 2006, 32nd International Conference on Current Trends in Theory and Practice of Computer Science , to appear*, January 2006.

[22] J. Gray, P. Homan, R. Obermarck, and H. Korth. A straw man analysis of probability of waiting and deadlock. In *Proceedings of the fifth Internafional Conference on Distributed Data Management and Computer Networks*, 1981.

[23] C.G. Harrison, D.M. Chess, and A. Kershenbaum. Mobile agents: Are they a good idea? IBM Research Report 19887, IBM Research Division, 1995.

[24] T. Herman and K. M. Chandy. A distributed procedure to detect and/or deadlock. Technical Report TR LCS-8301, Dept. of Computer Sciences, Univ. of Texas, 1983.

[25] Lisa Higham and Teresa Przytycka. A simple, efficient algorithm for maximum finding on rings. *Information Processing Letters*, 58:319–324, 1996.

[26] D. S. Hirschberg and J. B. Sinclair. Decentralized extrema-finding in circular configurations of processors. *Communications of the ACM*, 23(11):627–628, 1980.

[27] Richard C. Holt. Some deadlock properties of computer systems. *ACM Computing Survey*, 4(3):179–196, 1972.

[28] S. S. Isloor and T. A. Marsland. The deadlock problem: An overview. *IEEE Computer*, 13(9):58–78, Sep 1980.

[29] Mitsubishi Electric ITA. Mobile agent computing-a white paper. Technical report, Mitsubishi Electric Information Technology Center America, 1997.

[30] Kamal Jain, Mohammad T.i Hajiaghayi, and Kunal Talwar. The generalized deadlock resolution problem. In *Proceedings of the 32nd International Colloquium on Automata, Languages and Programming (ICALP)*, Lisboa, Portugal, July 2005.

[31] Ravi Jain, Farooq Anjum, and Amjad Umar. A comparison of mobile agent and client-server paradigms for information retrieval tasks in virtual enterprises. In *Proceedings. Academia/Industry Working Conference on Research Challenges*, pages 209–213, 2000.

[32] M. Jazayeri and W.A. Lugmayr. Gypsy: A component-based mobile agent system. In *Parallel and Distributed Processing, 2000. Proceedings. 8th Euromicro Workshop on*, pages 126–134, Jan 2000.

[33] Edgar Knapp. Deadlock detection in distributed databases. *ACM Comput. Surv.*, 19(4):303–328, 1987.

[34] Walter H. Kohler. A survey of techniques for synchronization and recovery in decentralized computer systems. *ACM Comput. Surv.*, 13(2):149–183, 1981.

[35] David Kotz, Robert Gray, Saurab Nog, Daniela Rus, Sumit Chawla, and George Cybenko. AGENT TCL: targeting the needs of mobile computers. *IEEE Internet Computing*, 1(4):58–67, July-Aug 1997.

[36] Natalija Krivokapic, Alfons Kemper, and Ehud Gudes. Deadlock detection in distributed database systems: A new algorithm and a comparative performance analysis. *VLDB J.*, 8(2):79–100, 1999.

[37] Ajay D. Kshemkalyani and Mukesh Singhal. On characterization and correctness of distributed deadlock detection. *Journal of Parallel and Distributed Computing*, 22(1):44–59, 1994.

[38] Gerard LeLann. Distributed systems – towards a formal approach. In B. Gilchrist, editor, *Information Processing 77*, pages 155–160. IFIP, North-Holland Publishing Company, 1977.

[39] Gertrude Neuman Levine. Defining deadlock. *SIGOPS Operating Systems Review*, 37(1):54–64, 2003.

[40] Vincenzo Della Mea. Agents acting and moving in healthcare scenario - a paradigm for telemedical collaboration. *IEEE Transactions on Information Technology in Biomedicine*, 5(1):10–13, March 2001.

[41] Toshimi Minoura. Deadlock avoidance revisited. *J. ACM*, 29(4):1023–1048, 1982.

[42] Abha Mittal and Sivarama P. Dandamudi. Dynamic versus static locking in real-time parallel database systems. In *Proceedings of 18th International Parallel and Distributed Processing Symposium.* IEEE, April 2004.

[43] H. S. Nwana. Software agents: An overview. *Knowledge Engineering Review*, 11(3):205–244, 1996.

[44] M. Overmars and N. Santoro. Improved bounds for electing a leader in a synchronous ring. *Algorithmica*, 18:246–262, 1997.

[45] G. L. Peterson. An $o(n \log n)$ unidirectional algorithm for the circular extrema problem. *ACM Transactions on Programming Languages and Systems*, 4:758–762, 1982.

[46] Stefan Pleisch and Andre Schiper. FATOMAS - a fault-tolerant mobile agent system based on the agent-dependent approach. In *Proceedings International Conference on Dependable Systems and Networks*, pages 215–224. IEEE Comput. Soc, July 2001.

[47] Charles M. Shub. A unified treatment of deadlock. *J. Comput. Small Coll.*, 19(1):194–204, 2003.

[48] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating System Concepts.* John Wiley & Sons, Inc., New York, NY, USA, 6th edition, 2001.

[49] Jonathan Smith. A survey of process migration mechanisms. *Operating Systems Review*, 1988.

[50] William Stallings. *Operating systems: internals and design principles.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 3rd edition, 1998.

[51] Todd Sundsted. An introduction to agents. *Java World*, June 1998.

[52] Andrew S. Tanenbaum. *Modern operating systems.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2001.

[53] Gerard Tel. *Introduction to Distributed Algorithms.* Cambridge University Press, 1994.

[54] Alexander Thomasian. Two-phase locking performance and its thrashing behavior. *ACM Trans. Database Syst.*, 18(4):579–625, 1993.

[55] J. Waldo. Mobile code, distributed computing, and agents. *IEEE Intelligent Systems*, 16(2):10–12, March-April 2001.

[56] Hong Wang, Guangzhou Zeng, and Shouxun Lin. A strong migration method of mobile agents based on java. In *Computer Supported Cooperative Work in Design, The Sixth International Conference on*, pages 313–318. IEEE, 2001.

[57] James E. White. Mobile agents. In Jeffrey M. Bradshaw, editor, *Software Agents*, chapter 19, pages 437–472. MIT Press, 1997.

[58] David Wong. Mobile agent development framework. Mitsubishi Electric ITA, Horizon Systems Laboratories.

[59] Dianlong Zhang. Network management using mobile agent. In X. Chunpei, editor, *Communication Technology Proceedings, 1998. ICCT '98. 1998 International Conference on*, volume 2, page 5, October 1998.