INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

Bell & Howell Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

UMI[®]

Distributed Deadlock Detection in Mobile Agent Systems

by

Bruce Ashfield

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment
of the requirement for the degree of

Master of Computer Science

**Ottawa–Carleton Institute of Computer Science
(OCICS)**

**Carleton University
Ottawa, Ontario
December 2000**

© Copyright

2000, Bruce Ashfield

National Library
of Canada

Acquisitions and
Bibliographic Services

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale
du Canada

Acquisitions et
services bibliographiques

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-57757-0

Canada

The undersigned recommend to the Faculty of Graduate Studies and Research
Acceptance of the thesis

Distributed Deadlock Detection in Mobile Agent Systems

Submitted by Bruce Ashfield
University of New Brunswick,
Bachelor of Computer Science, 1997

In partial fulfillment of the requirement for the degree of Master of Computer Science

Dr. Dwight Deugo, Thesis Supervisor

Dr. Frank K. Dehne, Director of School of Computer Science

Carleton University
January 01, 2001

Abstract

Mobile agent environments have unique properties and characteristics that represent a new application development paradigm. Generally speaking, many existing solutions to distributed computing problems, such as deadlock avoidance, are not suited to environments where both clients and servers move freely through the network. To support the assumptions of traditional solutions, restrictions or fundamental modifications to mobile agent systems must be enforced. New solutions and techniques must be developed and targeted for mobile agent systems to effectively and efficiently solve their problems.

This thesis suggests a distributed deadlock solution designed to function in a mobile agent system. The properties of this solution are locality of reference, network organization independence, fault tolerance, asynchronous execution and freedom of agent movement. The presented technique, called the shadow agent solution, suggests dedicated agents for deadlock initiation, detection and resolution. These agents are fully adapted to the properties of the mobile agent environment.

During the development of the solution, the properties and assumptions of traditional solutions are compared to the requirements of mobile agent systems. This analysis leads to a suggested set of properties that should be present in mobile agent solutions to distributed problems, such as deadlock detection. Finally, measurements for the suggested solution are presented, discussed and compared to those of similar traditional solutions.

Table of Contents

Abstract.....	III
List of Tables.....	VII
List of Figures.....	VIII
List of Appendices.....	IX
1.0 Introduction.....	1
1.1 Problem Statement.....	2
1.2 Goals and Objectives.....	3
1.3 Thesis Organization.....	4
2.0 Background.....	5
2.1 Deadlock Overview.....	5
2.2 Distributed System Coordination.....	8
2.3 Traditional Distributed Deadlock Detection.....	10
2.4 Software Agents.....	13
2.4.1 Agent Classification.....	14
2.4.2 Mobile Agents.....	17
2.4.3 Mobile Agent Advantages and Disadvantages.....	19
2.5 Mobile Agent Systems.....	22
2.5.1 Mobile Agent System Properties.....	22
2.5.2 Existing Mobile Agent Systems.....	24
2.6 Mobile Agent Coordination Techniques.....	25
2.6.1 Coordination Taxonomies.....	26
2.6.2 Coordination Models.....	28

2.7 Mobile Agent Influences on Traditional Solutions.....	32
2.7.1 Agent Movement.....	32
2.7.2 Failures and Fault Tolerance.....	34
2.8 Problem Summary.....	35
3.0 Mobile Agent Distributed Deadlock Solution.....	36
3.1 Mobile Agent Solution Properties.....	36
3.2 Deadlock Shadow Agent Technique.....	38
3.2.1 Premise and Assumptions.....	38
3.2.2 Algorithm Overview.....	41
3.2.3 Deadlock Initiation.....	44
3.2.4 Deadlock Detection	47
3.2.5 Deadlock Resolution.....	51
3.2.6 Deadlock Detection Example.....	54
3.2.7 Phantom Deadlocks.....	58
3.2.8 All Deadlocks Detected.....	60
3.3 Deadlock Detection Pseudo-Code.....	62
3.3.1 Consumer Agent.....	62
3.3.2 Shadow Agent.....	64
3.3.3 Deadlock Detection Agent.....	70
3.3.4 Host Environment.....	74
3.4 Algorithm Motivation and Agent Properties.....	78
3.5 Benefits and Challenges.....	81
3.6 Comparison of Proposed Solution to Traditional Solutions.....	84

3.7 Proposed Solution Theoretical Measurements.....	86
4.0 Mobile Agent Deadlock Modeling System.....	90
4.1 Simulator Platform Requirements.....	90
4.2 Simulator Implementation Overview.....	93
4.3 Simulator Architecture.....	94
5.0 Algorithm Measurements.....	100
5.1 Mobile Agent Solution.....	100
5.1.1 Measurement Methodology.....	101
5.1.2 Fault-Free Configuration Measurements.....	104
5.1.3 Fault-Injected Measurements.....	106
5.2 Measurement Discussion.....	109
5.3 Traditional System Measurements.....	112
6.0 Conclusions and Future Work.....	115
6.1 Conclusions.....	115
6.2 Summary of Contributions.....	117
6.3 Future Work.....	117
Bibliography.....	120

List of Tables

Table 1 Mandatory Agent Properties.....	15
Table 2 Secondary Agent Properties.....	15
Table 3 Mobile Agent System Functions.....	23
Table 4 Mobile Agent Systems.....	25
Table 5 Deadlock Detection Elements.....	43
Table 6 Deadlock Detection Information.....	48
Table 7 Resource Information.....	48
Table 8 Constructed Deadlock Information.....	57
Table 9 Simulation Environment Configurations.....	101
Table 10 Mesh Connected and Fault Free Measurements.....	105
Table 11 Ring Connected and Fault Free Measurements.....	105
Table 12 Mesh Connected and Failure Prone Measurements.....	108
Table 13 Ring Connected and Failure Prone Measurements.....	108

List of Figures

Figure 1 Mobility Taxonomy.....	18
Figure 2 Coordination Techniques.....	26
Figure 3 Deadlock Detection Information.....	51
Figure 4 Example Mobile Agent System.....	55
Figure 5 Simulator Interactions.....	95

List of Appendices

Simulator Execution Trace.....	124
--------------------------------	-----

1.0 Introduction

The increasing popularity of mobile agents as a form of application development often stresses or breaks traditional distributed computing techniques. Many existing solutions to problems, such as deadlock avoidance or leader election, are not suited to environments where both clients and servers move freely through the network. The dual movement of servers and clients creates interactions and new conditions for distributed applications that traditionally were not issues.

Forcing traditional distributed solutions into mobile agent systems changes the dynamic nature of these environments by imposing restrictions, such as limiting agent movement. The fundamental building blocks of many traditional distributed algorithms rely on assumptions such as data location, communication mechanisms or network organization (number of nodes, agent host locations and connections between hosts). The movement of both clients and servers in mobile agent systems breaks these fundamental assumptions. As a result, new solutions and techniques must be developed and targeted for mobile agent systems to effectively and efficiently solve their distributed coordination problems.

1.1 Problem Statement

In many distributed applications there are complex relationships between services and data. Mobile agents commonly encapsulate services and data (as do objects in object-oriented (OO) programming), which extends and enhances the data-service relationship by adding movement to both services and data. Traditional solutions to distributed problems tend break down in the dynamic and heterogeneous environment required to support mobile agent operations.

Generally speaking, the efforts of mobile agents must be coordinated to solve complex problems. Distributed deadlock detection is an example of the problems caused by the mobile agent paradigm. The advanced synchronization required in mobile agent based applications can lead to complex, multi node deadlock conditions that must be detected and resolved. Traditional distributed deadlock solutions fail when agent movement is added to traditional deadlock factors. Additionally, faults during the detection process must be considered when providing a detection scheme.

As mobile agents move through the network and interact with services, data and other agents, deadlocks may occur. These deadlocks must be detected to provide a robust, reliable environment. Due to their assumptions, traditional techniques such as edge chasing on the global wait-for graph, are inadequate solutions in a mobile agent system. Example mobile agent applications that require this solution are consensus, data transfer and distributed database processing.

This thesis develops a solution to the traditional problem of deadlock detection and resolution targeted for mobile agent systems. The presented solution is adapted to mobile agent environments through fault tolerance, asynchronous execution and absence of assumptions concerning the location or availability of data.

1.2 Goals and Objectives

The primary objective of this thesis is to develop a distributed deadlock detection technique adapted to mobile agent systems. The developed solution is not bound to a particular mobile agent environment or toolkit; therefore, is not tightly coupled or integrated with the mobile agents that implement a mobile agent application. As a result, properties specific to any mobile agent system do not influence the solution. An adapted solution must be tailored to the unique properties and challenges imposed by mobile agent environments. These challenges include increased fault frequency, autonomous consumer and resource movement, locality of reference and asynchronous communication.

A secondary goal of this thesis is to define the properties and qualities of a mobile agent solution to distributed problems. These properties, coupled with the developed deadlock detection technique, provide a distributed systems developer with a model for solving traditional problems in mobile environments. Efficiency and speed are secondary concerns in this model, but are considered and discussed when applicable.

To meet the stated objectives and goals of this thesis, a mobile agent distributed deadlock detection algorithm is presented. A platform independent simulation engine, which enforces a generic solution, is developed to verify the solution and gather measurements for the developed technique.

1.3 Thesis Organization

This thesis contains six chapters. Chapter two provides a general overview of mobile agents, their coordination, the unique properties of mobile agent systems and the problem of distributed deadlocks in mobile agent systems. Chapter three presents a technique for solving distributed deadlock detection in mobile agent environments. Chapter four discusses the simulation environment that was constructed to model and validate the proposed solution. Chapter five presents the measurement methodology implemented by the simulator. Chapter five also presents measurements for two network topologies, provides an analysis and compares these measurements to traditional distributed solutions. Finally, in chapter six conclusions are drawn and suggestions for future work are discussed.

2.0 Background

Since their introduction computer systems have managed resources that must be shared among competing entities or processes. As a result, a large body of research exists on subjects related to resource sharing, which includes deadlocks and their properties. This chapter introduces the basic properties of deadlocks in distributed systems. Additionally, a brief survey of traditional distributed deadlock detection techniques is presented.

Traditional solutions developed for distributed problem solving must be adapted when mobile agents are introduced. This chapter discusses the properties of mobile agents and mobile agent systems. Section 2.4 develops a working definition of a software agent and presents a possible agent classification scheme. Next, section 2.5 presents the properties of mobile agent systems using existing mobile agent systems as examples. Section 2.6 discusses various agent coordination techniques to illustrate the problems and complexities agent coordination introduces. Finally, mobile agent systems are compared to traditional distributed computing environments.

2.1 Deadlock Overview

Deadlock literature formally defines a deadlock as "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" [20][23]. A more informal description is that deadlocks can occur whenever two or more processes are competing for limited resources and the processes are allowed to acquire and hold a resource (obtain a lock). If a process waits for resources, any

resources it holds are unavailable to other processes. If a process is waiting on a resource that is held by another process, which is in turn waiting on one of its held resources, we have a deadlock. When a system attains this state, it is effectively dead and must resolve the problem to become operational.

Deadlocks are a direct result of the locking schemes which are required to share exclusive resources. There are four conditions that are required for a deadlock [23]:

1. Mutual exclusion. Each resource can only be assigned to exactly one resource.
2. Hold and wait. Processes can hold a resource and request more.
3. No preemption. Resources cannot be forcibly removed from a process.
4. Circular wait. There must be a circular chain of processes, each waiting for a resource held by the next member of the chain.

There are four techniques commonly employed to deal with deadlocks: ignore the problem, deadlock detection, deadlock prevention and deadlock avoidance [23].

Ignoring deadlocks is the easiest scheme to implement. This technique is acceptable in systems where the number of deadlocks is small compared to other fatal errors. Also, often in systems where ignoring deadlocks is acceptable, users cannot accept the restrictions and reduced flexibility that other deadlock techniques inherently must impose.

Deadlock detection attempts to locate and resolve deadlocks. This technique allows deadlocks to occur, detects their existence and takes actions to resolve the deadlock [20].

Strategies which fall into this category rely on a wait-for graph (WFG) that is built and analyzed for cycles. A cycle in the WFG represents a deadlock that must be resolved [23]. There are numerous techniques for both the detection and resolution of deadlocks. Depending on the requirements of the target system, any number of techniques are applicable, but regardless of the selected technique, system complexity is increased.

Deadlock avoidance describes techniques that attempt to determine if a deadlock will occur at the time a resource is requested and react to the request in a manner that avoids the deadlock. Deadlock avoidance techniques are commonly extensions of the banker's algorithm or are based on time stamps or transaction priorities [23]. The most significant challenge of techniques in this category is the gathering of detailed information at the time of the resource request. Often, sufficient information is not available, requiring a technique to predict the future if all deadlocks are to be avoided.

Deadlock prevention is the structuring of a system in such a manner that one of the four necessary conditions for deadlock cannot occur [20]. The disadvantages of techniques that fall into this category relate to the lack of flexibility or inefficiency that is introduced as a result. For example, if a process is required to request all the resources it may use before it begins to execute, other processes cannot use those resources, regardless of whether they will be used by the requesting process immediately or in the distant future. Restrictions such as this reduce concurrency and result in lower system efficiency. Finally, it is nearly impossible for all processes to know what resources they will require before they begin to execute.

Each solution category is suited to a specific type of environment and has advantages and disadvantages. This thesis focusses on deadlock detection and resolution, which is the most commonly implemented deadlock solution. The following sections discuss how the introduction of distributed transactions and process coordination deepen the problem of deadlock detection.

2.2 Distributed System Coordination

Coordination in distributed systems is related to how processes cooperate and synchronize their activities. In single processor systems, coordination is accomplished through traditional coordination mechanisms such as semaphores or monitors. Unfortunately, these techniques do not work well in the distributed domain since they commonly rely on the existence of shared memory [23].

In most distributed applications, a collection of processes share a set of resources that are managed by servers. The activities of the processes must be coordinated to ensure consistency, and as [20] suggests, "This is essentially the critical section problem, familiar in the domain of operating systems." There are many techniques available to synchronize activities in distributed systems, each suited to a particular application.

Clock synchronization plays a significant role in many distributed synchronization methods, because to make event related decisions in a distributed system, a global time is often needed [23]. Many clock synchronization algorithms have been proposed and

implemented, including logical clocks, the Berkly algorithm and clock averaging. A global time allows algorithms to be constructed based on agreement to coordinate entry into a critical section and is often called timestamping [20].

Distributed mutual exclusion is a common technique for distributed coordination and synchronization. There are two basic requirements for mutual exclusion: safety and liveness. Safety refers to the property that at most one process executes a critical section at a given time. Liveness is the property that any process requesting entry to a critical section is eventually allowed to execute that critical section. This second criteria implies that starvation and deadlock problems are avoided [20]. One category of mutual exclusion algorithms is based on centralized servers, but centralized techniques are not desirable for fault tolerance and scalability reasons. A second category of mutual exclusion algorithms are distributed by nature [23]. Techniques that fall into this category include ring based mutual exclusion and logical clock solutions [20].

Another coordination mechanism in distributed systems is based on an elected leader coordinating or sequencing activities. Techniques such as ring based leader election or the bully algorithm have been suggested to solve this problem [20]. Once elected, a process performs tasks such as master timer server, lock coordinator or event sequencer. To implement election algorithms the processes that make up a system must know the state and location of the other processes and must be able to uniquely identify each process [23]. In a large distributed system, this information is often hard to gather and maintain.

Atomic transactions are frequently employed in distributed systems to guarantee that either all the operations associated with a program unit are executed to completion, or none are performed. Implementing atomic transactions requires a transaction coordinator that is responsible for starting the transaction, breaking the transaction into sub-transactions, distributing the sub-transactions and coordinating termination [23].

The implementation of atomic transactions often requires a locking protocol to control access to shared resources. In general, two-phase locking protocols can be used in a distributed environment by changing how the lock manager is implemented [20]. Examples of locking protocols that modify the lock manager are: non-replicated, single-coordinator, majority protocol, biased protocol and primary copy. The implementation of locks can lead to deadlocks and other related problems, that must be considered when implementing a distributed system.

2.3 Traditional Distributed Deadlock Detection

Deadlocks in a distributed system have many properties similar to those found in single processor systems. Due to the lack of centralized information and resource control, they are harder to detect, avoid, prevent and resolve [23]. In distributed systems it is commonly the responsibility of servers to detect and resolve deadlocks. This simple statement represents a complex problem when dealing with networks of interconnected servers and transactions that are distributed across multiple servers.

As presented by [22], traditional distributed deadlock detection solutions can be divided into five categories: centralized, path–pushing, edge–chasing, diffusing computation, and global state detection. It is never trivial to detect and resolve deadlocks in a distributed system and many solutions have been proposed, each optimizing different system properties, such as the number of messages or frequency of detection. This is echoed by Tannenbaum [23] who observes "There are few areas of computer science in which theory and practice diverge as much as in distributed deadlock detection algorithms."

The simplest implementation of distributed deadlock detection is implemented using a centralized server to maintain the global wait–for graph. In this technique, periodically each server in the network sends to the central server its local copy of the wait–for graph, which is analyzed for deadlocks [20]. Obviously, this is both a fault intolerant and non scalable solution, since the single server can fail and must handle many transactions.

A better solution to this problem in distributed environments is to maintain local wait–for graphs and only construct the global graph when required. Another benefit of this approach is that the construction of the global graph has no central point of coordination. The categories based on this concept are path–pushing and edge–chasing [23].

Edge chasing, as described by [20], assumes that standard deadlock avoidance techniques, such as two phase transactions and resource locking, are employed. In this scheme, the servers in the network are aware of their peers and each connection represents a permanent edge in the global graph. Whenever a transaction begins waiting

on another transaction that is waiting on a resource on a remote server, a probe is initiated. This probe contains the contents of the server's local wait-for graph and is sent to the remote server that holds the resource on which the transaction is blocked. If necessary, the remote server will forward the graph to other servers. If a distributed deadlock is present, a cycle will be found in the global graph and action taken to break the deadlock. Path pushing techniques are similar, but differ in the manner that information is sent to neighbouring nodes [23].

The final two categories of distributed deadlock detection are diffusing computation and global state detection. Diffusing computation techniques are based on the principle of a spanning tree of the global wait-for graph representing the processes and their links. In systems organized such as this, processes are linked by their communication paths, but one process (the root process) is special in that initially only it can issue messages. Initially, any non-root processes cannot issue messages and must wait until they have received one. When applied to deadlock detection, the root process sends messages to its connected processes and eventually receives a message if a deadlock exists [23].

Algorithms falling into the global state detection category are based on similar concepts to edge-chasing, but order the system in a different manner to detect deadlocks. In this technique the global wait-for graph is constructed from local graphs without blocking the computation. The technique developed by [23] is based on this paradigm.

Detection schemes are commonly evaluated by two criteria [24]: 1) if an actual deadlock exists, it must be detected in a finite amount of time; 2) phantom deadlocks are not detected, i.e. the scheme must not find a deadlock that does not exist. Unfortunately, there is no universally accepted and deployed distributed deadlock solution, that is consistent with techniques used for single processor deadlock detection. Depending on the properties of an environment the two evaluation criteria must be applied and the most suitable scheme selected.

2.4 Software Agents

As [1] states, there is no universal definition of the term "software agent" or "agent". This lack of a single definition stems from factors such as the diverse and varied backgrounds of the groups working with agents (distributed artificial intelligence, decision theory, predicate logic) and the common nature of the word "agent".

Franklin and Graesser define an autonomous agent as "... a system situated within and part of an environment that senses that environment and acts on it, over time, in pursuit of its own agenda and so as to affect what it senses in the future." [2] Unfortunately, this type of definition leaves many applications on the fringe, and depending on the point of view, may or may not be perceived as an agent.

Franklin and Graesser also note that agent definitions are rarely of the same length or complexity and often appear to have grown out of descriptions of example agents [2].

This lack of a universal definition or sample set does not prevent common themes and attributes from being used to construct a working definition of a software agent.

For the purposes of this thesis, agents are defined by a subset of their attributes. The attributes considered to be a necessary part of an agent in the following sections are: activeness, autonomy, goal-driven and acting on behalf of a user or another agent [3]. There are many other attributes that play important roles in agents, but are not part of every agent. One notable attribute falling into this category is mobility.

Defining an agent by its essential attributes still leads to a large set of potential agents and requires further refinement. Any definition of agent should be considered a high level term or category that sits above a set of agent subclasses. One possible division of these agent subclasses is presented in section 2.4.1.

2.4.1 Agent Classification

In a similar fashion to the definition of agent, there are many attributes and properties that can be extracted from existing agents, but there is no precise set of attributes that can be used to classify agents. For example, as stated by [1] " ... mobility is neither a necessary nor sufficient condition for agenthood."

For this thesis, Table 1 [2][4] lists the properties that emerge from the necessary and sufficient attributes for agenthood that were discussed in section 2.4. Table 2 [2][4] lists a set of attributes that are of secondary importance that are not universally part of agents, but can be used as the basis of categorization schemes.

Table 1 Mandatory Agent Properties

Property	Description
Reactive	Senses and reacts to changes in its environment
Autonomous	Operates without the direct intervention of other (possibly human) agents
Goal Driven	Able to take the initiative to complete tasks
Temporally Continuous	Continually running

Table 2 Secondary Agent Properties

Property	Description
Communicative	Interacts and communicates with other agents
Mobile	Can move from host to host
Learning	Uses experience to adapt its actions
Believable	Appears believable to the user
Flexible	Not scripted

The number of different attributes that are commonly used to describe agents lead to many different classification schemes. For example, Brustoloni's taxonomy of software agents [5] divides agents into three major categories: regulation agents, planning agents and adaptive agents. This classification scheme is based on the amount of knowledge embedded in the agent.

Another common classification scheme is based on the internal representation of the world employed by an agent. Since the way an agent represents its world defines its behaviour, this division is also called mode of behaviour classification. Other classification schemes include task based, mobility based or any other attribute that is considered important by the classifier; therefore, no scheme is definitive or cannot be improved. As [1] states "In essence, agents exist in a truly multi-dimensional space, which is why we have not used a two or three-dimensional matrix to classify them – this would be incomplete and inaccurate."

For the purposes of this thesis, the categories suggested by [1] are used to divide agents into broad categories. This scheme defines seven agent categories: collaborative agents, interface agents, mobile agents, information / internet agents, reactive agents, hybrid agents and smart agents.

It is not the precise categorization of agents that is important to this thesis, but to note that there are many different types of agents that interact as part of an agent system. Additionally, it should be noted that agent categories are not mutually exclusive, but instead should be viewed as overlapping regions. For example, an agent that can be described as a collaborative agent, may also be a mobile and reactive agent.

In the context of this thesis, agents falling into the mobile category are of particular interest. The preceding discussion about categorization is important, since saying that an

agent is "mobile" does not describe how it operates. The way an agent completes tasks, interacts with other agents or represents the world, leads directly to how it behaves. An agent's behaviour is a significant factor in determining how susceptible it will be to traditional distributed computing problems, such as deadlock. Additionally, having an understanding of the types of agents available helps to develop more effective solutions.

2.4.2 Mobile Agents

Mobile abstractions are entities such as mobile code, objects and processes that allow the remote execution of instructions. Mobile agents represent the current evolutionary step in the abstraction of mobility [3]. The origin and idea behind mobile agents can be traced to the way computers have communicated since the late 1970's. The properties of remote procedure calls (RPC) and remote programming (RP) are the foundation for today's mobile agent technology. RPC defines a technique for computers to execute well defined procedures hosted by a remote machine. RP enhances this concept by not only defining the procedures that can be executed, but allowing the host to provide the procedure to be executed as well [6].

Early mobile software entities began to appear in the early 1980's in the form of worms that could arbitrarily move through the nodes of a network. The next step in the evolution of mobility was the invention of mobile processes at the operating system level. Although mobile processes are not widely deployed, they lead to the creation of mobile code that could be downloaded and executed on a remote machine (i.e., applets) [8].

Mobile agents are defined by [7] as active objects that have behaviour, state and location. A mobile agent's state consists of both code, data, thread and the authority of its owner. This property and the necessary attributes defined in section 2.4.1, separate mobile agents from mobile objects and processes whose states include fewer attributes.

Figure 1 **Mobility Taxonomy**

One possible mobile entity taxonomy has been proposed by [7] and is illustrated in Figure 1. The different types of mobility have their advantages and disadvantages, but this thesis is only concerned with the strongest form of mobility, agent mobility, and the challenges it poses to traditional distributed computing problems.

2.4.3 Mobile Agent Advantages and Disadvantages

As with any field of study, the mobile agent paradigm has advantages and disadvantages. Often these are described as benefits and challenges, or reasons for and against mobile agents. This section lists some of the common arguments for and against mobile agents.

Lange and Oshima [14] list seven arguments for mobile agents, which include:

- **Network load reduction.** This means that computations are moved to the data and less information is transferred across the network.
- **Network latency avoidance.** Network latency is reduced through centralized agent dispatch and local execution to allow quick execution of user commands.
- **Protocol encapsulation.** Agents contain all information related to a protocol and allow easy upgrading and implementation of proprietary protocols.
- **Asynchronous and autonomous execution.** Continually open network connections are not required to control mobile agents in error prone networks.
- **Adaptability and flexibility.** Since mobile agents sense and react to their environment, they are inherently adaptable and flexible.
- **Heterogeneous network tolerance.** Mobile agents are separated from platform specific issues; therefore, they can operate in a variety of environments.
- **Robust and fault-tolerant execution.** The dynamic adaptability of agents allows them to function in less than perfect conditions. For example, if a node fails, a mobile agent can leave the node and continue to execute elsewhere.

Additionally, as presented by [3] and [8] mobile agent systems have the following advantages:

- **Overcoming the limitations of a client computer.** Issues such as limited memory, computing power and storage space can be avoided by moving the mobile agent to a more suitable node.
- **Customization.** Searches or queries can be adjusted by user criteria or via the mobile agent detecting properties specific to the remote site.
- **Ease of development and the analogy to the real world.** Complex programming problems are often easier to understand when related to the physical world.

It should be noted that, the preceding advantages are not unique to the mobile agent paradigm, and can often be achieved through other techniques. For example, process migration, client server architectures or distributed protocols can yield many of the listed mobile agent advantages. It is the level of abstraction and infrastructure provided by mobile agents that makes these benefits attainable, without adding significant complexity. Additionally, it is rare that a single alternate technique can provide all of the advantages of mobile agents.

In the discussions of [3] [8] and [14], the following disadvantages of mobile agents are presented:

- **Lack of applications.** This is the most common disadvantage of mobile agent systems. There are many mobile agent implementations, but very few applications, giving mobile agents the reputation of "a solution searching for a problem" [8].

- **Lack of Standards.** The Object Management Group (OMG) Mobile Agent System Interoperability Facility (MASIF) project addresses this need, but has not been widely deployed. The lack of standards requires that applications have high development costs and risk being outdated by emerging standards.
- **Security.** Security for mobile code is not fully understood and mobile agents further extend this problem. This is an active area of research and has many undefined areas.
- **Lack of infrastructure.** Again, the MASIF project addresses this need, but has not been widely deployed. There is no common way of accepting, locating or describing agents, which hinders their deployment. Manual effort is often required to graft mobile agents to widespread technology such as web browsers.
- **Survivability.** This property is also listed as an advantage of mobile agents, but due to the complexity of embedding survivability into agents it is also a disadvantage or challenge of mobile agents.

Some of the applications that can benefit from the introduction of mobile agents are data mining, electronic commerce, software distribution, network management, personal assistance, groupware, parallel processing and system control [3] [8] [14]. As wireless networking and mobile computing capture more attention, mobile agent applications become attractive, since slow or unreliable links can benefit from mobile agent technology. Layer 7, sometimes called application switching, has been developed to efficiently move high level data through networks and the internet. These infrastructure devices can potentially be extended to host and support the intelligent movement of mobile agents.

2.5 Mobile Agent Systems

There are many mobile agent environments that are commercially or freely available. Each environment has its own advantages, disadvantages and unique properties. Currently there is no clear market leader or default standard, leaving the potential mobile agent developer with a difficult decision. In 1998 the OMG accepted the MASIF standard [9], but unfortunately it has not been widely deployed and accepted.

2.5.1 Mobile Agent System Properties

Mobile agent environment properties can include everything from the supported language(s), to the communication infrastructure or the security model. As a result, mobile agent environments can be sorted and classified in many different ways. Classifying and organizing mobile agents systems poses challenges similar to those of agent classification. It is not the intent of this thesis to provide a definitive classification of mobile agent environments, see the work by [12] or [13] for information on possible classification schemes.

The fundamental properties of a mobile agent system should provide support for the basic properties of an agent. In section 2.4.1, Table 1 and Table 2 enumerate these properties, which include: reactivity, autonomy, goal orientation, and temporal continuity. Additionally, a mobile agent environment must support agent properties such as mobility, learning and flexibility. Depending on the selected agent attributes, different mobile agent system descriptions are produced. For example, [12] defines a mobile agent

system as the following three tuple: "Mobile agent architecture = (Learning & Reasoning | Communication | Control Structure)".

Table 3 Mobile Agent System Functions

Property	Description	MASIF
Agent Transport	Also called Agent Migration. Transfer of agent state and type.	Yes
Agent Communication	Supported methods for agent to agent communication. Related to agent coordination, see section 2.6 for details. Types include asynchronous and synchronous.	No
Agent Tracking	Localization of agents in a distributed system, i.e., finding an agent.	Yes
Agent Management	Creation / Termination, Suspension / Resumption.	Yes
Agent Naming	Provide globally unique agent names and locations.	No
Control Structure	Also called constrained execution. Different from security. Types include: Hierarchy, Partially supervised, Hetrarchy.	No
Region Support	Set of agents that can be "found". Allows the mobile agent system to cooperate with other mobile agent systems.	No
Language Support	The source programming language used to express agents and their behaviour. Types include supported and partially supported.	No
Security	Mechanisms to preserve the privacy and integrity of data on a system. Standard techniques such as encryption, authentication, verification and authorization apply.	No

The preceding table summarizes the most common features of mobile agent environments. Some properties found in Table 3 have been standardized by MASIF [9], while others have been added based on the surveyed mobile agent environments and the research of [12] and [13].

The mobile agent application developer is primarily concerned with the features and properties of particular environments. In the end, each project must decide which features and abstractions are part of their selection matrix, rate the available environments, and choose the most appropriate mobile agent system.

2.5.2 Existing Mobile Agent Systems

The selection of a mobile agent environment influences the ability of the mobile agent developer to deal with traditional distributed computing problems (i.e., deadlock detection). The synchronization, messaging, control and other primitives provided by the mobile agent system can potentially increase the possibility of common distributed computing problems. The following table lists some of the many mobile agent development platforms that are currently available.

Table 4 Mobile Agent Systems

Name	Language(s)	Coordination Techniques
Mozart [18]	Derivative of OZ-3	Direct, Meeting Oriented
Telescript [6]	Telescript	Direct, Meeting Oriented
Aglets [4][25]	Java	Direct
Agent TCL (d'Agents) [26][27]	Tcl/Tk, Java, Scheme, C, C++	Direct
Concordia [28][29]	Java	Direct, Meeting Oriented, Blackboard
Mole [7][30]	Java	Direct, Meeting Oriented
TACOMA [31]	Tcl, C, Python, Scheme, Perl	Direct, Blackboard
Ara [21]	Tcl/Tk, Java, C, C++	Direct, Meeting Oriented
MOA [8]	Java	Direct, Meeting Oriented
Voyager [32][33]	Java	Direct, Meeting Oriented
ffMAIN [34]	Tcl, Perl, Java	Direct, Blackboard
Messengers [35]	M0	Blackboard
Odyssey [36]	Java	Direct, Meeting Oriented
MARS [37]	Java	Direct, Linda-Like

2.6 Mobile Agent Coordination Techniques

Distributed applications have complex data-service relationships. For example, an application may have servers and client processes that execute or reside on physically separate network environments. In distributed situations such as this, the client and server must communicate via messages. Mobile agents introduce new factors, such as client and server movement, when they are added to a distributed environment. These new factors further complicate the already difficult task of remote communication;

therefore, traditional communication and coordination techniques must be extended to support mobile agents.

All of the applications suggested in section 2.4.3 as potentially benefitting from mobile agent technology require the cooperation of multiple mobile agents. The cooperation of mobile agents requires that the resources and primitives provided by the mobile agent system be used to coordinate their activities. This section describes mobile agent coordination taxonomies and models that can be used to control cooperating mobile agents.

2.6.1 Coordination Taxonomies

Figure 2, taken from [11] describes a mobile agent coordination taxonomy based on the kinds of coupling introduced by coordination schemes.

Figure 2 **Coordination Techniques**

		Temporal	
		Coupled	Uncoupled
Spatial	Coupled	Direct	Blackboard Based
	Uncoupled	Meeting Oriented	Linda-Like

This classification represents the extension of existing coordination model research to emphasize the types of coupling found in mobile agent systems. The properties used to divide the various coordination schemes are the degree of spatial and temporal coupling incurred by the technique.

The elements of mobile agent coordination are directly related to the agent naming and agent communication properties introduced in section 2.5.1. In other words, the ability of agents to coordinate their activities depends on the naming and communication primitives supplied by the mobile agent system. As stated by [1], "Coordination is essential to enabling groups of agents to solve problems effectively. Without a clear theory of coordination, anarchy or deadlock can set in easily in collaborative agent systems."

Agent coordination and agent computation should be recognized as orthogonal properties of agent systems to provide a separation of the behaviour of an agent and the way it interacts with its environment [15]. This statement implies that mobile agent coordination must consider the internal and external interactions of mobile agents as they cooperate to complete tasks.

Another coordination taxonomy is suggested by [12]. This model divides agent coordination into synchronous, federated and asynchronous communication and has many of the same properties and coupling concepts as those found in Figure 2. Finally,

techniques such as coordination without communication [19] must also be considered and do not easily fit into any taxonomy. The classification of [11] is favoured in this thesis, due to its explicit references to temporal and spatial coupling.

2.6.2 Coordination Models

Figure 2 illustrates two major types of coupling that limit the flexibility and adaptability of a mobile agent system. Temporal coupling refers to the execution order of mobile agents, or more precisely, examines if their communication needs to be synchronized. Spatial coupling refers to the existence of a shared name space required to allow agents to communicate. This name space ranges from message formats to agent names and locations [11].

Direct communication is the most rigid form of mobile agent coordination, since it enforces spatial and temporal coupling. In this coordination mechanism, both agents must be executing on the same server if they wish to communicate. If communicating mobile agents are not executing on the same host, then complex routing protocols (i.e., agent proxies as in Aglets [4]) are required to make the network transactions transparent. This form of communication requires constant network connections; therefore, its efficiency and fault tolerance suffer. Agents whose activities are coordinated with this mechanism must know the names and / or location of their peers in order to establish communication [11]. This form of coordination is equivalent to the mobile agent / service agent interactions defined by the Mole project [7]. In mobile agent / service

agent interactions a client server arrangement is required, making direct coordination a necessary property.

Meeting oriented coordination limits spatial coupling by removing the necessity of location and name sharing before communication. In this mechanism, the meeting names must be either implicitly or explicitly shared, but do not represent spatial coupling. Temporal coupling remains an issue since agents may only communicate with peers that have joined the same meeting. Meetings must be local to a node or environment or their implementation becomes complex as remote message passing is required to connect the agents participating in the meeting. This coordination technique results in less network traffic than direct communication [11] and is commonly found in mobile agent systems such as Mole [7] or embedded in operating systems [16]. Meeting oriented coordination represents a form of mobile agent to mobile agent interaction that must be described and controlled by the mobile agent hosts.

Blackboard based coordination limits temporal coupling by providing a shared data space where agents can deposit messages for other agents. This is a fully asynchronous communication technique, since an agent may create a message, leave the server and the recipient may arrive and retrieve the message at a later time [11]. As [12] presents, asynchronous architectures supporting blackboard based coordination are characterized by common communication places for all mobile agents in the system. All agents are aware of these places and communicate with each other indirectly through the shared data space. These shared data spaces remove the requirement of meeting or agent names

for communication. Spatial coupling remains an issue since the agents using the blackboard to communicate and coordinate must agree on a message format. This technique creates a dynamic agent system and provides a great deal of control to the local environment [11]. This form of coordination can be mapped to the anonymous agent group interaction defined by Mole [7].

The most flexible technique illustrated in Figure 2 is Linda-Like coordination. As described by [17], this communication technique provides a shared, globally accessible, persistent tuple space. This scheme is similar to blackboard based communication in the sense that it employs a shared data space, called a tuple space, to provide inter-agent communication. The difference between Linda-Like coordination and blackboard based coordination is the method of accessing the shared data space. In Linda-Like coordination, some sort of associative technique is provided to access messages, as opposed to the more rigid message formats of blackboards. An example associative technique is regular expression matching. Associative techniques allow agents to communicate without defining the format of the messages, by searching on common strings and fragments [11]. The combination of asynchronous communication and associative message retrieval provides both temporal and spatial decoupling. As a result, Linda-Like coordination can adapt to a mobile, dynamic and heterogeneous environment.

Even though Linda-Like coordination provides a great deal of flexibility and leaves room for extension (see reactive tuple spaces) it is not always the best solution for mobile

agent coordination. Depending on requirements such as efficiency, network congestion, mobility level and communication format, one of the other techniques can be more appropriate and easier to implement. Regardless of the selected mobile agent communication mechanism, distributed problems such as deadlock detection or leader election must still be solved. The selected agent communication scheme may facilitate or provide mechanisms to implement a solution, but do not solve all distributed problems in isolation.

Many mobile agent environments and operating systems provide primitives for at least one of the presented communication schemes. For example, the Mozart Programming System [18] has direct support for three of the four techniques (Direct, Meeting Oriented and Blackboard based) but with its multi-paradigm support it can easily be extended to provide Linda-Like coordination.

The following is a small list of mobile agent systems that support the discussed coordination mechanisms:

- Direct Communication: Aglets [4][25], D'Agents [26][27], Mozart [18]
- Meeting Oriented Communication: ARA [21], Mole [7][30], Mozart [18]
- Blackboard Based Communication: Ambit [38], ffMain [34], Mozart [18]
- Linda-Like Communication: PageSpace [39], MARS [37]

2.7 Mobile Agent Influences on Traditional Solutions

This section presents how the properties of mobile agent systems influence traditional distributed problem solving techniques. Traditional distributed techniques such as distributed shared memory or distributed barriers can be adapted to work in mobile agent systems, but impose restrictions or inefficiencies that are often unacceptable to mobile agent applications. Special attention is paid to distributed deadlock detection, but the problems and their solutions are applicable to a wider range of problems.

Traditional distributed deadlock detection solutions commonly have location and fault assumptions. The following two sections compare the properties of mobile agent systems and traditional distributed computing environments that cause these assumptions to fail. In particular, agent movement and failure recovery are discussed.

2.7.1 Agent Movement

In traditional distributed deadlock detection (section 2.3) resources and their consumers do not move through the network. Instead, distributed transactions are initiated locally and access data items located on remote nodes. Distributed transactions can be simple or nested, but regardless of their complexity, standard coordination techniques are employed. The techniques discussed in section 2.2 require each server to have knowledge about the location of the nodes that form the network.

In a mobile agent system, agents perform transactions by moving towards the source of data and executing locally to leverage the benefits of locality of reference [8]. The

mobile agent and the host server can continue to interact with additional network resources [6] and as a result, transactions can be spread across multiple hosts without informing the node that initiated the transaction. Clearly, agent movement causes problems for algorithms that depend on location information.

In distributed deadlock detection techniques, such as central server or edge chasing [22], location assumptions cannot be avoided, since information is either gathered centrally or built through a series of probes. In order to detect and resolve distributed deadlocks, these algorithms must be able to locate the nodes involved in the transaction [20]. In a mobile agent system, an agent's movement and activities cannot be easily tracked; therefore, the agent that initiated a transaction can be difficult to locate, much less all of the secondary agents that are indirectly involved.

Assumptions concerning location must be adapted if an algorithm is to function properly and efficiently in a mobile agent system. It is not desirable to fix this problem through over synchronization of agent activities, since this restricts flexibility and the survivability of the application. Additionally, relying on a complex proxy or forwarding environment exposes the agent environment to failures and network delays, effectively removing two of the benefits of mobile agents. It should be assumed that mobile agents are self-contained and can freely move through the network based on routing instructions or via self-discovery.

2.7.2 Failures and Fault Tolerance

Traditionally, distributed problem solving techniques have not dealt well with failures or crashes. For example, the distributed deadlock detection algorithm described by [20] states "The above algorithm should find any deadlock that occurs, provided that waiting transactions do not abort and there are no failures such as lost messages or servers crashing."

The nature of mobile agents and their operating environment is failure prone; therefore, algorithms with strict fault assumptions will have difficulty functioning in many mobile agent systems. The chances of mobile agent failures and faults are increased by the inherent properties of the mobile agent system which require agents to navigate a large and unreliable network (such as the internet). Any number of failures ranging from host crashes to communications failures can easily trap or crash an agent [21]. One of the common reasons for mobile agent deployment is slow or unreliable links. In environments with these links, predictable behaviour and successful communication become suspect.

The combination of the fault assumptions of traditional techniques with the failure prone nature of mobile agent systems illustrates the need to adapt existing solutions for mobile agent applications. It is not a requirement that mobile agent solutions to problems, such as distributed deadlock detection, can seamlessly handle catastrophic errors, but survivability must be emphasized and failures expected. Fault tolerance is not only the responsibility of the mobile application developer, it is aided by the properties of the

mobile agent environment. For example, the "checkpoint" functionality provided by Ara [21] allows application developers to prepare for, and recover from, crashes and failures.

2.8 Problem Summary

As discussed in section 2.6, mobile agents must coordinate their activities to complete tasks and achieve common goals. Interacting mobile agents have dynamic relationships that change with time and adapt to their environment. This provides great freedom, but dictates that the application designer cannot depend on finding a server or client in the same location on subsequent accesses.

Traditional distributed deadlock detection techniques cannot be easily placed into a mobile agent environment and mobile agents cannot be easily squeezed into an existing solution without causing failures, inefficiency or other problems. Traditional distributed deadlock detection should be modified and adapted to function properly in a mobile agent system.

For the purposes of distributed deadlock detection, a mobile agent system is viewed as a group of asynchronous, but dependent, entities playing the roles of both clients and servers. In such a system, there are few restrictions on the movement of the mobile agents to promote maximum flexibility. These mobile agents are cooperating and sharing resources that often can only be accessed by a single agent. In this scenario, distributed deadlock is a danger and must be avoided.

3.0 Mobile Agent Distributed Deadlock Solution

This chapter presents a solution to distributed deadlock detection in mobile agent systems. First, the properties of mobile agent techniques are generalized and presented. Next, the proposed solution is developed and discussed in detail. The detailed description is followed by its motivation and properties. Next, the benefits and challenges of this solution are discussed. Finally, the presented solution is contrasted and compared to traditional distributed deadlock detection techniques.

3.1 Mobile Agent Solution Properties

The solution to distributed deadlock detection proposed in this thesis contains features that are suited to the dynamic environment in which mobile agents are commonly found. These features separate this solution from existing solutions and illustrate why traditional solutions are inappropriate for a mobile agent environment. The features of a mobile agent solution to this problem fall into four major categories: network organization independence (number of nodes, location of nodes and inter-node connections), agent movement, fault tolerance and limited coupling between the solution and the mobile agent system.

Network organization independence, which can also be called topology independence, refers to the property that the deadlock detection algorithm should not depend on a particular organization of the nodes in a mobile agent network. This property represents a logical organization of the nodes, and includes the number of nodes in the network,

their location and the connections between the nodes. This is a logical property, since in both traditional distributed solutions and mobile agent solutions, software is not aware of the physical connections between nodes and the routing that must be performed during message passing. This property means that agents can move at will throughout the network even as its topology changes. This allows advanced properties such as dynamic network updates and node swapping to be performed without affecting the success of the algorithm. Network organization independence must be part of the mobile agent hosts and the mobile agents themselves. Many traditional solutions depend on specific network organizations, such as a ring, to detect and resolve deadlocks.

Agent movement should not be unnecessarily restricted by the deadlock detection algorithm. For example, agents must be allowed to lock resources and move to other hosts to perform additional tasks. The decision to block is made by the agent alone. The host environment is informed of the agent's blocked state to allow necessary deadlock detection steps to be taken. This property makes it difficult to fit a traditional solution into a mobile agent environment, since the location of transactions and servers must be known. This is not possible in a mobile agent environment without restricting when and how agents can move.

Fault tolerance is an important property of a mobile agent solution, since the chances of failure are increased as agent movement is introduced. This property is implemented through autonomous / asynchronous agent execution, exploiting locality of reference and fault detection and recovery. Distributed deadlock detection in a mobile agent

environment should assume that messages or steps will be lost and gracefully recover when they occur. Many traditional deadlock detection algorithms assume that messages are not lost and that communication is reliable.

The mobile agents that consume and manipulate resources in the agent system should be separated from the deadlock detection and resolution process. This allows the algorithm to function in numerous systems and mobile agent development environments. To achieve this goal, agents communicate with their host environments through well defined Application Programmer Interfaces (APIs) and deadlock analysis is performed in reaction to these calls without the agent's knowledge.

3.2 Deadlock Shadow Agent Technique

The following sections describe the solution to distributed deadlock detection in mobile agent environments proposed by this thesis. The foundation and assumptions of this algorithm are presented first, followed by the detection and resolution phases of the algorithm. Finally, an example deadlock detection process is described in section 3.2.6.

3.2.1 Premise and Assumptions

In section 3.1 the desired properties of a mobile agent solution to distributed deadlock detection are presented. These properties are the foundation for the assumptions behind this solution to distributed deadlock detection in a mobile agent system.

It is assumed that all forms of mobile agents are separated from the topology of the network, which means they can move through the network without knowledge of how many nodes there are or how they are connected. This movement is accomplished via node or resource based routing and the agent requests to be routed to a particular environment or resource. It is up to the host environment to route the agent on a best-effort basis. This technique is similar to the routing scheme used in Internet Protocol (IP) routers.

It is assumed that the topology of the network is static once the algorithm begins. If the topology is allowed to dynamically update, a Mobile Ad-hoc Networks (MANET) type topology update protocol must execute in the background to keep routing tables synchronized with the actual topology.

The mobile agent distributed deadlock algorithm in this thesis assumes that standard deadlock avoidance techniques, such as two phase commit or priority transactions, are being used. These mechanisms allow the detection and resolution process to be assured that an agent will not spontaneously unlock a resource (and unblock) during the detection process. This property is a significant factor in preventing phantom deadlock detection.

The solution presented by this thesis assumes that resources can only be locked and unlocked by a consumer agent when it is physically at the same environment as the resource it is manipulating. This property allows host environments to communicate the

details of a consumers agent's resource requests to its associated deadlock detection counterparts.

In order for deadlocks to occur, it is assumed that there is a level of cooperation between agents or shared resources. As the agents perform their tasks, resources can be "locked", which means they are made exclusive to an individual consumer agent (i.e., single writer, many reader problem).

Additionally, it is assumed that during the locking process, consumer agents must communicate with the host environment. The host is the ultimate authority and can allow or deny access to a resource. Since the host environment can deny the lock request of an agent, a reaction is required. It is assumed that depending on the agent's task, it may block or wait on the resource (and will not move) or it may continue processing and moving through the network. It is assumed that the agent is not automatically blocked by the authority, since this would limit flexibility and limit the dynamic nature of the mobile agent environment.

This solution assumes that agents must notify the host environment if they block on the resource. This allows the environment to communicate the state of an agent to its deadlock detection counterparts and deny any additional requests made by the blocked agent. Blocked agents cannot unblock until their request is approved by the host environment.

Agents must be uniquely identifiable once they hold a resource. This means that they can be found in the agent system during the deadlock detection process. 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).

3.2.2 Algorithm Overview

This mobile agent algorithm uses agent-adapted mechanisms that are based on those found in traditional edge-pushing global wait-for graph algorithms [20]. In particular, the division of the global wait-for graph into locally maintained segments and the initiation of deadlock detection "probes" are inspired by traditional solutions.

There are three types of agents populating the mobile agent system: consumer agents, shadow agents and detection agents. In this solution, shadow agents maintain a part of the wait-for graph on behalf of their target consumer agent. During the deadlock detection process the shadow agents continually check for deadlocks in the global wait-for graph and create deadlock detection agents to build the global graph. Detector agents are small reactive agents and are used instead of probes in traditional edge-pushing solutions.

The combination of host environments, consumer, shadow and detection agents cooperate to detect and resolve any deadlocks that occur in the mobile agent system. The following sections describe the phases of the algorithm, which are initiation, detection

and resolution. The final section describes the advantages of this technique in a mobile agent environment.

The elements and their tasks that are part of this distributed deadlock detection solution are described in the following table.

Table 5 Deadlock Detection Elements

Element Name	Description
Consumer Agent	<ul style="list-style-type: none"> • The only agents in the system that actively perform tasks and consume resources. • Only agent that can lock resources. • Represent the entities which perform tasks and implement algorithms. • No active role in deadlock detection and resolution.
Shadow Agent	<ul style="list-style-type: none"> • Created by host environments. • Responsible for maintaining the resources locked by a particular consumer agent and following it through the network. • Responsible for initiating the deadlock detection phase when informed that their target agent has blocked. • Responsible for analyzing the data gathered by detection agents and initiating deadlock resolution if required. • Responsible for detecting and recovering from faults during the deadlock detection process. • Represent a portion of the global wait-for graph.
Detection Agent	<ul style="list-style-type: none"> • Created by shadow agents when informed by the host environment that their target agent has blocked. • Small lightweight mobile agents • Responsible for visiting hosts and building the global wait-for graph. • Responsible for breaking the deadlock situation.
Host Environment	<ul style="list-style-type: none"> • Responsible for hosting the mobile agents. • Provide APIs for consumer and shadow agents to access resources, move, block, unblock, etc. • Coordinates the consumer, shadow and detector agents as they move through the network. • Coordination is performed through a "token" that is carried by all consumer agents.

3.2.3 Deadlock Initiation

As consumer agents complete tasks, they may exclusively lock resources throughout the mobile agent system. When consumer agents are initially created, they are not actively monitored by the host environments for deadlock detection reasons. This means that new agents can freely move through the network and consume resources. 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.

Consumer agent monitoring activities begin when an exclusive lock is requested by an agent. As part of the request granting process, the host environment checks for a shadow agent that is associated with the requesting agent. If no shadow exists, one is created and associated with the consumer agent. Finally, the consumer agent's environment token is updated to reflect the existence of the newly created shadow agent. Once a shadow agent is created for a consumer agent, the host environments are able to control the deadlock detection process.

Shadow agents are notified of new exclusive locks by host environments through a defined message. This message includes deadlock detection information, such as the identifier and priority of the locked resource. Resource locking is performed through a defined API; therefore, the host environment always has a chance to update shadow

agent information before granting the request. Each new lock is noted by the shadow agent and maintained as part of its deadlock detection information.

The mobile agent system's wait-for graph can be represented by the notation: $C_x [R_1, R_2, \dots, R_n] \rightarrow R_b$. In this notation, C_x represents a consumer agent that has exclusively locked resources. C_x can be replaced by S_x to more accurately represent the actual mobile agent system, since shadow agents are created to monitor consumer agents. The values $[R_1, R_2, \dots, R_n]$ are a form of distributed state and represent the resources that are held exclusively by a particular consumer agent. Finally, the R_b notation following the arrow ($\rightarrow$) represents the resource on which an agent is blocked. If a consumer agent is not blocked, the $\rightarrow R_b$ portion of the notation is not present. Combining all of the local states (represented by this notation), creates the global wait-for graph for the mobile agent system.

Once a shadow agent is created and associated with a consumer agent, they move through the network together. This synchronous movement is coordinated by automatically routing a consumer's shadow each time the consumer sends a migration request to the host environment. It should be noted that this pairing of agents places restrictions on consumer agents. If a consumer agent's shadow is not present, a consumer agent cannot migrate, lock or unlock a resource. A shadow agent may be absent due to network latency, congestion, transmission failure or other errors.

Consumer agents are notified of failed requests and must resubmit to retry failed transactions. Shadow agents must be aware of all the lock and unlock transactions performed by their target agents to maintain the exact state of the wait-for graph. This information is passed locally from host environments to shadow agents; therefore, shadow agents must be in the same host as their target (during resource transactions) to maintain the wait-for graph. This is guaranteed, since resource lock and unlock transactions only succeed if the required shadow agent is present and are retried until the shadow arrives. This restriction has minimal impact during execution, since the delay of shadow agents is rare and only occurs in exceptional situations. Section 6.3 discusses an alternative organization of shadow and consumer agents to eliminate this assumption.

When a consumer agent requests an exclusive lock that is denied by a host environment, it may decide to block and wait for the resource to be freed. If the decision to block is made, the consumer agent must inform the host environment. Host environments react to blocking notifications by informing the consumer agent's shadow to allow deadlock information to be checked. It should be noted that if the consumer holds no locks, a shadow agent will not be present and cannot be informed. This non-notification is acceptable, because if the consumer agent holds no other locks, it cannot be part of a distributed deadlock.

Shadow agents are informed by the host environment that their target has blocked or unblocked through a message. These block and unblock events begin the deadlock

initiation process. When informed of a block event, shadow agents query the host environment to determine who holds the lock on the target resource. Once the host environment provides the agent identifier who holds the lock, a second query is performed to determine if the agent is local or remote. If the locking agent is off-site, the distributed deadlock detection sequence is initiated by the shadow agent, otherwise, no special processing occurs.

3.2.4 Deadlock Detection

Shadow agents initiate the deadlock detection cycle by creating detection agents. Upon creation, detection agents are initialized with their parent shadow agent's list of locked resources and the environments where they are located. This creation of a dedicated detection agent allows a shadow to simultaneously search for deadlocks and respond to other shadow's detectors.

Once initialized, detector agents visit the resources that are locked by their target consumer agent. By recording the network location of each resource as it is locked, routing of detector agents is expedited. At each resource the detector agents visit, they query the host environment to determine if other agents are blocked on that resource. If blocked agents are found, the detector locates their associated shadow agent and queries for their deadlock detection information. This processing occurs simultaneously for every agent that is blocked on a resource held by a remote agent. The returned deadlock detection information is a list of deadlock detection entries that are summarized in the following tables.

Table 6 Deadlock Detection Information

Agent Name	The unique identifier of the consumer agent that this information belongs to.
Blocked Resource	The resource information for the resource that the agent is blocked on. See Table 7 for more information.
Primary Locks	The list of primary locks (resources) held by this agent. These are in resource information format, see Table 7 for more information.

Table 7 Resource Information

Resource Name	The unique name of the resource.
Resource Owner	The consumer agent that has this resource locked.
Environment Name	The environment name that contains this resource.
Priority	The priority of this resource.

Each deadlock detection entry encapsulates the relevant information about a consumer agent that is blocked on a resource. At each resource, this information is added to the detector's original detection table, increasing the amount of information it carries as it moves through the network. The deadlock information of each blocked agent is added to the detector's deadlock detection table because the agent is blocked on a resource held by the detector's target. Since these agents are blocked on a resource held by another agent, their entire detection table (and hence any agents blocked on them) is indirectly being held by that agent.

This secondary deadlock information is valid, since blocked agents cannot move to free resources while waiting for the resource locked by a detector's target. When a detector

agent visits all of the resources that were placed in its initial set of locks, it returns to its initial host environment. 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.

Shadow agents use their target consumer agent as a deadlock detection key. If their target agent appears in the deadlock information table returned by the detector, the target is waiting on a resource held by itself. Clearly we have a deadlock (a cycle in the global wait-for graph), since the target is blocked and that resource will never be freed.

Shadow agents are responsible for recovering from failures during the deadlock detection phase. Failure detection is implemented through a running round trip delay calculation. Depending on the type of network and its properties, each shadow agent is initialized with a default round trip delay time. Shadow agents expect that their detector agents will be able to check all of their required locks in less than four times the maximum round trip delay. If a detector agent does not return in the maximum allowable time, the shadow agent assumes that it has failed and creates a new detector agent to continue the failed agent's work.

Each time a detector agent returns from checking locks its parent shadow agent updates its expected round trip delay time. This allows the shadow agents to continually tune their expectations to current network conditions. The detector agents created by a

shadow are versioned to prevent errors if a shadow incorrectly assumes that its detector agent is dead. Each time a detector agent is created, the shadow agent notes the version number and will only accept deadlock information from that shadow agent. If old detector agents eventually return, they are killed and their information ignored.

The failure of a detector agent has a direct impact on the efficiency of the algorithm, since information is lost and must be recovered. The impact of failures is minimized by the information summarizing performed by shadow agents. When a detector agent fails, only its current information is lost and the data from previous trips is safe; therefore, only a single visit must be repeated.

If a shadow agent does not find a deadlock in the global wait-for graph, the detector agent is re-initialized and the process repeats. Depending on the requirements of the implementation and the properties of the mobile agent system, a delay may be desired between detector visits. This allows the algorithm to be tuned between implementations and to react to network conditions. Possible delay techniques include: no delay, delay for a configured duration (i.e. one second), delay for a random duration or delay for a duration based on the round trip time. Delaying based on the round trip is an example of using network conditions to provide continual feedback and adjust the algorithm accordingly.

Once a shadow agent finds a deadlock in the global wait-for graph, the deadlock resolution sequence is initiated. Section 3.2.5 discusses this phase of the solution.

3.2.5 Deadlock Resolution

This section discusses how the deadlock detection information gathered in section 3.2.4 is used to resolve a deadlock. When a shadow agent detects a deadlock, there is a cycle in the global wait-for graph that must be broken. To successfully resolve the deadlock the shadow agent must identify the cycle and determine how to break the deadlock.

The information gathered by the detection agents is sufficient to build the deadlock cycle and determine which resource should be unlocked. The following figure illustrates how the deadlock detection table represents a graph.

Figure 3 Deadlock Detection Information

The location of the deadlock cycle is facilitated by the entrance criteria to the resolution phase. These criteria dictate that there is a cycle in the graph, and the repeated resource is

the resource on which the shadow is blocked. The detection table creates the detection graph as follows:

1. The resource on which the detecting agent is blocked is the root node in the detection graph. This is test.db in Figure 3.
2. The primary locks held in the root detection table entry form the next nodes in the graph. This is account.db in Figure 3.
3. As each primary lock entry is added to the graph, the entries of agents that are blocked on that resource are added as children of that node. These are user.db and foo.db in Figure 3.
4. Step 3 repeats for each child node as it is added to the graph, until there are no additional entries to be added or the root node's resource is located.

Note that the graph illustrated in Figure 3 is not normally constructed and it is sufficient to search a table-based data structure to determine the deadlock cycle. To determine the deadlock cycle, the shadow agent must first locate the leaf node that represents the resource on which its target consumer is blocked. The leaf nodes in Figure 3 are representations of the primary locks held by each blocked agent; therefore, the primary lock lists held in the detection tables are searched to locate the leaf node. Once that node has been located, the shadow agent must simply walk the graph to the root node (which is also the blocked resource) and record each node on the way. The information gathered for each node in the graph is sufficient to locate the deadlock and make a resolution decision.

When a shadow agent detects a deadlock, it has the authority to break the deadlock. One of the easiest deadlock resolution techniques to implement is based on transaction priorities. Using this technique, when each node is added to the mobile agent system it is assigned a priority identifier. The priority of a 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 a 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.

If a deadlock exists, all agents that are part of the deadlock will detect the situation and try to initiate resolution. As each shadow agent detects a deadlock and builds the deadlock cycle, it checks the priorities of all the resources that are part of the cycle. It selects the resource that has the lowest priority as the one to be unlocked. Once the selection has been made (each agent has selected the same resource), the shadow agents check to see if they are the owner of that resource, i.e., if they have it locked.

The shadow agent that owns the resource that must be unlocked is responsible for breaking the deadlock. Shadow agents break deadlocks by instructing their detector

agents to travel to the deadlock resource and unlock it. Additionally, the shadow provides the name of the consumer agent that should be notified when the resource is unlocked. This ensures that in situations where multiple agents are blocked on a single resource the consumer agent that is part of the deadlock is notified and unblocked.

Once initialized, the detector agent travels to the correct host environment and requests that the resource be unlocked. If the deadlock resource is successfully unlocked, the detector agent returns to its shadow while the host environment notifies the specified consumer agent. When the shadow returns to its parent agent, the shadow removes the resource from its primary lock set and destroys the detector.

The final step of the deadlock resolution is carried out by the host environment. When the host environment notices that a resource which has blocked agents is unlocked, it notifies the consumer agent specified by the detector agent that the resource has been freed. The unblocked consumer agent can then complete its lock request and the deadlock is broken.

3.2.6 Deadlock Detection Example

The following section illustrates the deadlock detection process described in the preceding sections. In this example mobile agent system there are four environments E_1 , E_2 , E_3 and E_4 and their priorities are 1, 2, 3 and 4 respectively. In this configuration 1 is the highest priority and 4 is the lowest. E_1 hosts a single resource R_1 and two agents A_1

and A_2 . E_2 hosts resource R_2 and agent A_3 . Finally, environments E_3 and E_4 contain resources R_3 and R_4 respectively.

Figure 4 Example Mobile Agent System

The mobile agent system begins in an unlocked, freely moving state where agents A_1 , A_2 and A_3 are performing tasks on behalf of off-line users. The deadlock detection process begins when the following sequence of events occur.

1. A_1 locks R_1 . Shadow S_1 is created.
2. A_3 locks R_2 . Shadow S_3 is created.
3. A_3 moves to E_1 .
4. A_1 moves to E_4 .
5. A_1 locks R_4 .
6. A_1 moves to E_2 .

7. A_1 attempts to lock R_2 and blocks.

8. A_3 attempts to lock R_1 and blocks.

At this point, the mobile agent systems wait-for graph is represented by $S_1 [R_1, R_4] \rightarrow R_2$ and $S_3 [R_2] \rightarrow R_1$, where S_1 and S_3 are the shadow agents created to monitor consumer agents A_1 and A_3 .

At steps five and six in the previous list, S_1 and S_3 detect that the resource on which their target blocked is being held by an off-site agent. This initiates the deadlock detection process:

1. S_1 creates a detector agent D_1 .
2. S_3 creates a detector agent D_3 .
3. D_1 periodically visits E_1 and gathers the deadlock detection tables for any agents blocked on R_1 . Note that the period between visits is tuned to the requirements of the particular solution.
4. D_3 periodically visits E_3 and gathers the deadlock detection tables for any agents blocked on R_2 . Note that the period between visits is tuned to the requirements of the particular solution.

This repeating process results in the following deadlock detection tables being created by D_1 and D_3 . The values contained in parenthesis represent the resource information described in section 3.2.4 and are: resource name, owner agent, environment, priority.

Table 8 Constructed Deadlock Information

Agent Name	Blocked On Resource	Primary Locks
A_1	$R_2 (R_2, A_3, E_2, 2)$	$R_1 (R_1, A_1, E_1, 1), R_4 (R_4, A_1, E_4, 4)$
A_3	$R_1 (R_1, A_1, E_1, 1)$	$R_2 (R_2, A_3, E_2, 2)$

When D_1 and D_3 return to their shadow agents (S_1 and S_3) their constructed tables are analyzed for possible deadlocks before being added to the shadow's local detection information. Each shadow scans for its target agent in the returned table to recognize a deadlock situation. In this example, each shadow agent detects a deadlock, since their target agents appear in the returned tables on the first iteration.

The resolution phase is now started. Each shadow determines the deadlock cycle and the lowest priority resource in the cycle. In this case, the resources involved in the cycle are determined to be R_1 and R_2 . Even though both shadow agents find that R_2 has the lowest priority and must be unlocked, only S_3 owns R_2 and is responsible for breaking the deadlock. The following events occur to break the deadlock and allow processing to continue:

1. S_3 instructs D_3 to travel to E_2 and unlock resource R_2 and notify A_1 when unlocked.
2. D_3 moves to E_2 .
3. D_3 unlocks E_2 .
4. D_3 leaves E_2 to notify its shadow of success.
5. E_2 notifies A_1 that the resource it wanted is now free.
6. A_1 locks R_2 and continues processing.
7. D_3 arrives at E_1 and notifies its shadow that R_2 is no longer locked.
8. A_3 performs recovery from having its resource unlocked.

This simple example illustrates the fundamental properties of this mobile agent distributed deadlock detection technique. More detailed examples were used when validating the solution and are measured in chapter 5.

3.2.7 Phantom Deadlocks

The assumptions stated in section 3.2.1 dictate that deadlocks cannot be spontaneously resolved. More precisely, agents cannot spontaneously free a lock and unblock, thereby breaking a potential deadlock cycle. Additionally, each cycle in the global wait-for graph represents a deadlock. The global wait-for graph is maintained collectively by the shadow agents and constructed via the activities of detector agents.

These properties limit the number of ways that a phantom deadlock can be detected. A phantom deadlock could be detected if either of the following conditions occur: 1) a

single deadlock cycle is independently resolved by multiple agents or 2) if the resolution of a deadlock breaks another deadlock without the knowledge of the agents involved in that deadlock.

In the shadow agent technique, a cycle in the global wait-for graph is never independently detected by multiple shadow agents. Each shadow agent is the only entity in the mobile agent system with a complete list of the locks held by its consumer agent. Shadow agents represent all of the edges in the global wait-for graph that terminate on the resources held by that agent. These can be viewed as multiple incoming edges in the wait-for graph. 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.

A cycle in the global wait-for graph can only be detected through cooperation between shadow agents. All of the shadow agents involved in a deadlock detect the deadlock condition using exactly the same information. This information includes sufficient details to determine which resource should be unlocked, which agent should unlock the resource and which agent should be notified when that resource is unlocked. As a result, there will always be a single shadow agent that is responsible for resolving the deadlock, preventing condition number one, as described above, from occurring.

The assumed resource locking model requires that at a particular time no more than one agent may hold an exclusive lock on a resource and an agent may only block on a single

resource. Additionally, shared locks are not part of the global wait-for graph since they do not cause agents to block when access is requested. As was previously discussed, a blocked agent represents a single outgoing edge in the global wait-for graph and its locked resources represent multiple incoming edges.

Due to the single outgoing edge property of shadow agents, there is only one cycle that can be found in the global wait-for graph for that agent. This property means that agents and resources cannot be in multiple cycles in the global wait-for graph; therefore, another deadlock resolution process cannot unlock one of the resources in the cycle. This means that condition two, as described above, will not occur.

Based on the assumptions and properties of the presented technique, this discussion shows that neither condition one nor condition two can occur. The absence of these conditions means that phantom deadlocks will not be detected by the suggested solution.

3.2.8 All Deadlocks Detected

In order to demonstrate that the suggested solution finds all deadlocks, several observations are required. First, it should be noted that outgoing edges in the global wait-for graph occur only when an agent blocks on a resource. As each agent blocks, it must notify its current host environment. It is the responsibility of that host environment to notify the consumer agent's related shadow agent. The result is that all outgoing edges in the global wait-for graph are reported to a shadow agent.

Secondly, whenever a consumer agent locks a resource this information is reported to its associated shadow agent or the lock request will fail. As a result, shadow agents represent all of the incoming edges of the global wait-for graph that reference a consumer agent's locked resources. A single shadow agent manages all of the incoming and outgoing edges in the global wait-for graph for a single consumer agent and is the authoritative reference for this information.

The presented mobile agent solution requires that each shadow agent be able to locate all of the resources locked by its target consumer agent. The location of these resources is used by detector agents when gathering deadlock information. This allows a shadow agent to guarantee that it can gather details concerning the set of incoming edges it represents.

If an incoming edge (as viewed by a shadow agent) is connected to the global wait-for graph, there must be outgoing edges from other agents leading to the resource held by the shadow agent. To check for these outgoing edges, detector agents visit the set of locked (incoming) resources of their shadow agent. Detector agents perform repeated visits to build the global wait-for graph. Eventually, detector agents are guaranteed to locate all of the outgoing edges that directly or indirectly reference their locked (incoming) resources. This is true because all resource locking must be performed locally; therefore, detector agents are guaranteed to locate all outgoing edges that reference a particular resource by visiting that resource. If a deadlock is present, a shadow agent will locate its own outgoing edge referencing its locked (incoming) resource.

To summarize, a shadow agent is guaranteed to eventually locate all of the outgoing edges that reference its locked resources. A deadlock is represented by the existence of an agent's outgoing edge in the set of edges gathered by its detector agent. Therefore, it follows that if a deadlock exists, it will be detected. At the time a shadow agent detects the deadlock all of the information required to break the deadlock has been gathered. As a result, it also follows that all detected deadlocks will be resolved in a finite amount of time.

3.3 Deadlock Detection Pseudo-Code

This section presents pseudo-code of each element that plays a significant role in the presented solution. This skeleton code provides a context for the solution and insight into the responsibilities and activities performed by each entity. First, pseudo-code for the consumer, shadow and detection agent is presented. Finally, code for the mobile agent environment is presented.

3.3.1 Consumer Agent

```
public class ConsumerAgent extends MobileAgent
{
    public ConsumerAgent( String int heartbeat )
    {
        state = IDLE;
    }

    public void run()
    {
        while( true )
        {
            messages = getMessagesFromBlackboard( agentId );
            processMessages( messages );
        }
    }
}
```

```

switch( state )
{
    case IDLE:
    case WAITING:
        // do nothing
        break;
    case MOVING:
        if( currentHost is not targetEnvironment ) )
        {
            postRouteRequest( targetEnvironment );
        }
        else
        {
            // Made it!
            state = IDLE;
        }
        break;
}
sleep( heartbeatDelay );
}
}

private Vector processMessages( messages )
{
    while( more Messages )
    {
        processMessage( currentMessage );

        if message was accepted remove from list;
    }
    return unprocessed messages;
}

private boolean processMessage( BlackboardEntry msg )
{
    if( message equals "move" AND state is IDLE )
    {
        targetEnvironment = get target from message;
        state = MOVING;

        // lets ask our current environment to route us
        postBlackboardMsg( "route",
                           targetEnvironment );
    }
    else if( message equals "lock" AND
             state is IDLE OR WAITING )
    {
        extract lock type and resource from message ;
        lockResource( lockType, resource );
    }
    else if( message equals "unlock" AND state is IDLE )
    {
        extract resource from message;
        unlockResource( resource );
    }
}

private void lockResource( String lockType,
                           String resourceName )

```

```

{
    if( lockType equals "exclusive" )
    {
        get resource manager;

        if resourceManager.lockResource( resourceName,
                                         lockType ) succeeds
        {
            // locked. the AgentEnvironment should have created
            // a shadow agent and placed us under it's watchful
            // eye.
            // Special case. if the resource we just locked was
            // the same as a resource we were blocked on, it
            // means the Environment notified us and we should
            // move into the idle state
            if( resourceName equals blockedResourceName )
            {
                state = IDLE;
            }
        }
        else
        {
            // locked failed .. time to block
            state = WAITING;
            blockedResourceName = resourceName;

            // Notify our gracious host ...
            postBlackboardMessage( "agentBlock",
                                  resourceName );
        }
    }
}

private void unlockResource( String resourceName )
{
    get Resource Manager

    resourceManager.unlockResource( resourceName );
}
}

```

3.3.2 Shadow Agent

```

public class ShadowAgent extends MobileAgent
{
    public ShadowAgent( String id,
                       String targetAgent,
                       int heartbeat )
    {
        state = IDLE;
        targetAgentName = targetAgent;

        reset locked resource list
        reset detection info table
        reset detector agent
    }

    public void run()
    {

```

```

while( true )
{
    // Check to see if our detector is dead
    checkForDetectorDeath();

    if( state is not MOVING )
    {
        messages = getMessagesFromBlackboard();
        messages = processMessages( messages );
    }

    switch( state )
    {
        case IDLE:
            break;
        case MOVING:
            if( currentHost is not targetEnvironment )
            {
                routeRequest( targetEnvironment );
            }
            else
            {
                state = IDLE;
            }
            break;
    }

    sleep( heartbeatDelay );
}

private Vector processMessages( messages )
{
    while( more Messages )
    {
        processMessage( currentMessage );
        if message processed remove from list;
    }

    return unprocessed messages;
}

private boolean processMessage( BlackboardEntry msg )
{
    Vector attachments = msg.getAttachments();

    if( message equals "move" AND state is IDLE )
    {
        String target = extract target from message;

        targetEnvironment = target;
        state = MOVING;
    }
    else if( message equals "addLock" )
    {
        addLock( attachment #1,
                  attachment #2,
                  attachment #3,
                  attachment #4 );
        retVal = true;
    }
}

```

```

    }
    else if( message equals "removeLock" )
    {
        removeLock( attachment #1,
                     attachment #2 );
        state = IDLE;
        retVal = true;
    }
    else if( message equals "blocked" ) )
    {
        blockedTarget( attachment #1 );
        retVal = true;
    }
    else if( message equals "unblocked" ) )
    {
        unblockedTarget();
        retVal = true;
    }
    else if( message equals "deadlockReport" ) )
    {
        processReturnOfDetector( attachment #1,
                                 attachment #2,
                                 attachment #3 );
        retVal = true;
    }
    else if( message equals "deadlockInfoRequest" ) )
    {
        postBlackboardMessage( "deadlockRequestResponse"
                               copyDetectionTable() );

        retVal = true;
    }
    else
    {
        // We don't understand this message, but our superclass
        // might have some good ideas ...
        retVal = super.processMessage( msg );
    }
}

public void exit()
{
    // If our detector agent isn't null it means that our
    // target agent was blocked. It has now been allowed to
    // move (hence the exit() call as we are following.
    // This means the deadlock is broken and we should kill
    // our detector and re-create it later if needed.
    if( detector )
    {
        remove detector from host environment;
    }
    super.exit();
}

private void addLock( String environment,
                     String resource,
                     String owner,
                     int priority )
{

```

```

        // Check to see if we contain this lock already
        if( resource not already locked )
        {
            // construct an information object
            new resourceInfo( env, res, owner, priority );
            store resourceInfo in locked resource list
        }
    }

private void removeLock( String env, String resourceName )
{
    if( resourceName is in locked resource list )
    {
        remove resource from locked resource list
    }
}

private void unblockedTarget()
{
    // Our target has just unblocked
    state = IDLE;
}

private void blockedTarget( Agent blockedAgent,
                           String resourceName )
{
    state = TARGET_BLOCKED;
    owner = query host environment for owner of resource;
    blockedResourceName = resourceName;

    // Is the owning agent here ?
    localAgents = query host environment about local agents;

    if( owner in localAgents )
    {
        // Possible deadlock start the algorithm!
        // 0) Make an entry in our detection table about the
        //     resource we are blocked on.
        detectionInfoTable.put( targetAgentName,
                               new DetectionInfo( .. ) );

        // 1) Create a detector agent
        detector = new DetectorAgent();

        // 2) Inject the agent
        postBlackboardMsg( "inject",
                          detector );

        // 3) start the agent
        postBlackboardMsg( "start",
                          buildDetectorLocks() );

        numOfDetectionStarts++;
        lastDetectionStartTime = current time;
    }
}

private void processReturnOfDetector( DetectorAgent agent )
{

```

```

// Our detector is back, let's see what's new
switch( state )
{
    case TARGET_BLOCKED:
        if( checkForDeadlock( agent.getDetectionTables() ) )
        {
            // We have a deadlock, better resolve it.
            resolveDeadlock( agent );

            reset detectionInfoTable;
        }
    case IDLE:
        // Our target unblocked if this is the case ...
        // let's kill the detector ...
        removeDetector();
}

// The state will have changed to waiting for unlock
// if the deadlock check succeeded
switch( state )
{
    case TARGET_BLOCKED:
        // reset and restart the detector
        postBlackboardMsg( "start",
                           buildDetectorLocks() );

        numOfDetectionStarts++;
        lastDetectionStartTime = current time;

        break;
    case WAITING_FOR_UNLOCK:
        // We are breaking the deadlock don't start
        // any new processing.
        break;
}
}

private boolean checkForDeadlock( Vector detectionTableList )
{
    // - If we find the resource that our master is blocked
    //   on in the returned lock list ... we have a
    //   deadlock
    // - If we don't find the resource, just add the locks
    //   to our global list.

    // Let's go through the returned table .. entry by entry
    // and add it to our detection table.

    while( detectionTableList has more entries )
    {
        detectionTable = current detection table;
        agentList = get agent list from detectionTable;

        while( agentList has more entries )
        {
            detectionInfo = detection info from current table
                           related to current agent;
            add detectionInfo to global table;

            if( current agent name equals targetAgentName ) )

```



```

        {
            deadlockFound = True;
        }
    }
}
return deadlockFound;
}

private void resolveDeadlock( DetectorAgent agent )
{
    // Build a list of the resources involved in the cycle ...
    if( state is TARGET_BLOCKED )
    {
        // So we found a deadlock .. the question is are
        // we the one to break it ?
        // Find the Cycle ...
        cycleList = findElementsInCycle( detectionInfoTable );

        while( cycleList has more elements )
        {
            // find resource with lowest priority
            lockToBreak = lowest priority resource;
        }

        if( lockToBreak equals resource we are blocked on )
        {
            // Let's send our detector off on his mission
            // to unlock the resource ..
            // But first we better set him up with the
            // correct information to survive the
            // destination ResourceManager's interrogation.

            // 3) start the detector agent
            postBlackboardMsg( "unlock",
                               lockToBreak );

            state = WAITING_FOR_UNLOCK;
        }
    }
}

private Vector findCycle( Hashtable detectionInfoTable )
{
    Vector cycleVector = new Vector();

    cycleVector add( resource we are blocked on );

    info = find entry in detectionInfoTable whose primary
            lock is the current resource;

    // Loop until we find the entry for our agent
    while( current entry's agent name isn't equal to
           our target agent )
    {
        // Walk up the tree to the parent node.
        info = find entry in detectionInfoTable whose primary
                lock is the current resource;
    }
}

```

```

        cycleVector add( info );
    }
    return cycleVector;
}

private void checkForDetectorDeath()
{
    Date currentTime = current time;
    BlackboardEntry msg;

    if( numOfDetectionStarts > 0 AND
        ( state is TARGET_BLOCKED OR
          state is WAITING_FOR_UNLOCK ) )
    {
        if( ( currentTime.getTime() -
              lastDetectionStartTime ) >
            ( waitFactor * roundTripAvg ) )
        {
            // We have a dead detector
            // 1) Create a detector agent
            detector = new DetectorAgent();

            // 2) Inject the agent
            postBlackboardMsg( "inject",
                              detector );

            if( state is TARGET_BLOCKED )
            {
                // 3) start the agent
                postBlackboardMsg( "start",
                                  buildDetectorLocks() );
            }
            else if( state is WAITING_FOR_UNLOCK )
            {
                resolveDeadlock( detector.getIdentifier(),
                                detector.getToken() );
            }

            lastDetectionStartTime = current time;
        }
    }
}

```

3.3.3 Deadlock Detection Agent

```

public class DetectorAgent extends MobileAgent
{
    public DetectorAgent( String id, int heartbeat,
                          ShadowAgent parent )
    {
        reset detection Table List;
        reset resources To Vist;
        reset targetEnvironment;
        reset targetResource;

        state = IDLE;
        set parent = parent;
    }
}

```

```

}

public void run()
{
    while( true )
    {
        if( state is not MOVING )
        {
            messages = getMessagesFromBlackboard();
            messages = processMessages( messages );
        }

        switch( state )
        {
            case IDLE:
                break;
            case MOVING:
                if( currentHost is not targetEnvironment ) )
                {
                    routeRequest( targetEnvironment );
                }
                else
                {
                    state = CHECKING_LOCKS;
                }
                break;
            case UNLOCKING:
                if( currentHost is not targetEnvironment )
                {
                    routeRequest( targetEnvironment );
                }
                else
                {
                    // We've arrived ... let's break the lock!
                    if( host.unlockResource( targetResource,
                                            agentToNotify ) )
                    {
                        state = RETURN_FROM_UNLOCK;
                    }
                    else
                    {
                        state = IDLE;
                    }
                }
            }
            case RETURN_FROM_UNLOCK:
                if( currentHost is not startingEnvironment )
                {
                    routeRequest( startingEnvironment );
                }
                else
                {
                    // Made it back .. let the shadow agent know
                    // we did it!
                    Shadow.removeLock( targetEnvironment,
                                      targetResource );

                    state = IDLE;
                }
            }
            case DONE:
                if( currentHost is not startingEnvironment )
                {

```

```

        routeRequest( startingEnvironment );
    }
    else
    {
        // Made it back ..
        state = REPORT_RESULTS;
    }
    case CHECKING_LOCKS:
        checkLocks();
        break;
    case REPORT_RESULTS:
        postMessageToBlackboard( shadowAgent,
                                deadlockInfo );
        break;
    }
    sleep( heartbeatDelay );
}

private Vector processMessages( messages )
{
    while( more Messages )
    {
        processMessage( currentMessage );
        if message processed remove from list;
    }
    return unprocessed messages;
}

private boolean processMessage( BlackboardEntry msg )
{
    attachments = msg.getAttachments();

    if( message equals "start" ) )
    {
        startDetection( (Vector) attachment #1 );
        retVal = true;
    }
    else if( message equals "unlock" ) )
    {
        startUnlock( attachment #1,
                    attachment #2,
                    attachment #3 );
        retVal = true;
    }
    else if( message equals "deadlockRequestResponse" ) )
    {
        deadlockRequestResponse( attachment #1 );
        retVal = true;
    }
    else
    {
        super.processMessage( msg );
    }

    return retVal;
}

private void startDetection( resources )
{

```

```

    setVisitList( resources );

    // We haven't started yet ...
    ResourceInfo entry = get first resource to visit;

    // Let's get started ...
    targetEnvironment( entry.getEnvName() );
    targetResource( entry.getResName() );

    // Reset the table ...
    detectionTableList( new Vector() );
    startingEnvironment( getHost().getName() );
    state( MOVING );
}

private void checkLocks()
{
    // Ask the current host for any blocked agents ..
    // - We are in since we hold the lock for that resource
    //   there is a possibility of a distributed deadlock.

    shadowList = query current host for agents blocked on
                  the resource we are visiting;

    // Lets enumerate through the returned agents and get
    // their lock lists
    while( shadowList has more elements )
    {
        postBlackboardMsg( "deadlockInfoRequest",
                           currentShadow );

        count expected responses;
    }

    if( expected responses > 0 )
    {
        state = WAITING_FOR_RESPONSE;
    }
    else
    {
        findNewTarget();
    }
}

private void deadlockRequestResponse( newTable )
{
    expectedResponses--;
    detectionTableList.add( newTable );

    if( all expectedResponses received )
    {
        // Done!
        findNewTarget();
    }
}

private void findNewTarget()
{
    if( more resource to visit )
    {

```

```

        get next resource;

        // Let's get started ...
        targetEnvironment = entry.getEnvName();
        targetResource = entry.getResName();
    }
    else
    {
        // Time to head home ..
        targetEnvironment = startingEnvironment;
        state = DONE;
    }
}
}

```

3.3.4 Host Environment

```

public class AgentEnvironment extends Thread
{
    public AgentEnvironment( String name,
                             int id,
                             int loggingLevel )
    {

        resourceManager = new ResourceManager();
        topologyManager = new TopologyManager();

        reset agentTable;
        reset messageBoard;
        reset blockedAgentTable;

        // Global Identifier can be used as the priority
        globalIdentifier = id;
        state = PROCESSING;
    }

    public void run()
    {
        while( true )
        {
            checkForMessages();
            updateRoutes();
            sleep( 1000 );
        }
    }

    public synchronized void agentEnter( Agent newAgent )
    {
        if( state is PROCESSING )
        {
            agentTable.put( newAgent );
            newAgent.enter();
        }
    }

    private synchronized void agentExit( Agent leavingAgent )
    {
        if( state is PROCESSING )
        {

```

```

        leavingAgent.exit();
    }
}

private void agentBlock( Agent blockedAgent,
                        String resourceName )
{
    // Look for it's shadow agent ...
    shadow = find shadow agent for blockedAgent;
    if( shadow found )
    {
        // Tell it that it's agent has blocked. This may
        // start the deadlock detection process ...
        postBlackboardMsg( "blockedAgent",
                           resourceName );
    }

    blockedAgentTable.add( blockedAgent );
}

private void checkForMessages()
{
    get messages from blackboard;

    while( more messages )
    {
        processMessage( current message );
    }
}

private void processMessage( BlackboardEntry msg )
{
    attachments = msg.getAttachments();

    if( message equals "pause" ) )
    {
        state( PAUSED );
    }
    else if( message equals "resume" ) )
    {
        state( PROCESSING );
    }
    else if( message equals ( "block" ) ) )
    {
        agentBlock( msg.getAgentId(),
                    attachment #1,
                    attachment #2 );
    }
    else if( message equals "route" ) )
    {
        routeRequest( msg.getAgentId(),
                     attachment #1,
                     attachment #2 );
    }
    else if( message equals "inject" ) )
    {
        this.injectAgent( attachment #1 );
    }
    else if( message equals "remove" ) )
    {

```

```

        removeAgent( attachment #1 );
    }
}

private boolean routeRequest( String movingAgent,
                             EnvironmentToken token,
                             String targetEnv )
{
    if( state() is PROCESSING )
    {
        // Get the requesting agent ...
        movingAgent = get moving agent from agent tables;

        if( movingAgent is null )
        {
            // Might have been a duplicate post ...
            return true;
        }

        // Check to see if the agent is allowed to move ..
        if( check for shadow information in the token )
        {
            shadowAgentId = get shadow name from token;

            // This agent has a shadow agent, is the shadow here ?
            If( check for shadow agent in agent tables )
            {
                // The shadow is here .. everything is fine.
                shadow = get shadow agent from agent tables;
            }
            else
            {
                // No Shadow ... no move
                retVal = false;
            }
        }

        if( retVal is true )
        {
            AgentEnvironment env = request route from
                                   topologyManager;

            if( env is not null )
            {
                suspendAgent( movingAgent );
                agentExit( movingAgent );
                agentTable().remove( movingAgentId );
                env.agentEnter( movingAgent );

                if( shadow is not null )
                {
                    suspendAgent( shadow );
                    agentExit( shadow );
                    agentTable().remove( shadow );
                    env.agentEnter( shadow );
                }
            }
            else

```



```

        {
            retVal = false;
        }
    }
else
{
    // Not processing ...
    retVal = false;
}

return retVal;
}

public Vector getBlockedAgents( String resourceName )
{
    return list of agents blocked on resourceName;
}

public synchronized void postMessage( String agentId,
                                       String message )
{
    add message for agentId to the message lists;
}

public synchronized Enumeration getMessages( String agentId )
{
    return messages for agentId;
}

private injectAgent( Agent newAgent )
{
    // A user has enqueued a new Agent ...
    // start the agent running (since it is a thread)
    // and store it in our table
    newAgent.start();
    agentTable.put( newAgent );

    agentEnter( newAgent );
}

private void removeAgent( String agentName )
{
    agentTable.remove( agentName );
}

private void updateRoutes()
{
    topologyManager.updateRoutes();
}
}

```

3.4 Algorithm Motivation and Agent Properties

The presented algorithm is designed with the unique properties and challenges of mobile agent systems as a motivating factor. As a result, the solution has some of the properties and features that are commonly found in mobile agent implementations. This section lists the properties of the proposed algorithm which make it a mobile agent solution.

The solution is network organization independent. The algorithm makes no assumptions concerning network topology (i.e., ring), the number of hosts or node locations to support the solution. Resource-based routing and tracking of the nodes visited by a particular agent eliminate the need for explicit topology knowledge. Network organization independence is guaranteed through a clear separation of mobile agents from the mechanics of routing and migration. Additionally, the agents are not aware of the number of hosts in the mobile agent system and do not have explicit knowledge of resource locations. It should be noted that even though the solution is network independent, the topology is static once the algorithm begins. If the topology is allowed to change, a dynamic topology update protocol must execute in the background to provide new routes to the hosts.

The presented technique is fault tolerant and robust. Lost agents or messages during the deadlock detection process do not represent a critical failure. This fault tolerance is due to three properties of the algorithm: the autonomous nature of the agents, the periodic nature of the detection process and the copying of deadlock information. Shadow, deadlock detection and consumer agents execute asynchronously and do not depend on

continual communication during the deadlock detection process. The algorithm is designed around incremental construction of the global wait-for graph; therefore, if a portion of the graph is lost, the next update will recover that information. Finally, the copying of the partial wait-for graph into deadlock detection agents make the loss or failure of a particular deadlock detection agent trivial and has no impact on the detection process, outside of slowing the process. Additional safeguards can be built into the agent hosts, such as agent crash detection, to improve fault tolerance.

A common use of mobile agents is to encapsulate complex protocols and interactions [14]. This technique uses the combination of shadow agents and deadlock detection agents to encapsulate a complex series of probes, interactions and acknowledgments. Additionally, these protocols are isolated from the consumer agent; therefore, can be easily modified and upgraded. The deadlock detection phase could be implemented as remote procedure calls or another form of distributed programming, but would require network organization assumptions and the continual exchange of messages.

Detector and shadow agents carry out their deadlock detection tasks in an asynchronous manner. They coordinate their efforts in defined ways, but are able to keep working without regular contact and do not require constant supervision while carrying out tasks. This asynchronous and autonomous operation contributes to the previously discussed fault tolerance.

The combination of consumer, shadow and detector agents adapt to their environment to solve deadlock situations. For example, shadow agents react independently to changing network conditions and the state of their target consumer agent to initiate the deadlock detection processing. Similarly, detector agents can react to network failures or the requests of other agents while gathering global wait-for graph information.

The separation of the implementation from facilities specific to a particular mobile agent system or operating system allows the solution to execute in a heterogeneous environment. Additionally, the separation of shadow and detector agents from the consumer agents they monitor, allows them to be adapted to many different environments without (or with minor) modifications to the entities performing the work.

The presented solution leverages locality of reference during the deadlock detection process by migrating detector agents to query multiple blocked agents. An alternative to this single migration, is to message each blocked agent individually and gather their responses at the shadow agent itself. Detector migration reduces network load by limiting the number of messages that would be required in other solutions. It is difficult to compare the network load of this mobile agent solution to that generated in traditional distributed deadlock detection solutions due to the significantly different paradigm and properties of the environment.

If the network load of the presented solution is considered as a whole, the load is spread across many host environments. This is due to the parallel / distributed nature of the

technique, which enforces the lack of a central point of messaging and coordination. This reduces the risk of flash congestion and allows the technique to handle deadlock involving many blocked agents.

As was previously presented, traditional distributed solutions commonly have fault and location assumptions that make them unsuitable for mobile agent systems. To solve this problem, mobile agent specific solutions are required. The properties of the presented deadlock detection algorithm illustrate how it is a fully adapted mobile agent solution.

3.5 Benefits and Challenges

Security and reliability are two of the most significant problems addressed by mobile agent technology [3]. As a result, many of the inherent benefits of mobile agent solutions are based on solutions to these problems. The presented mobile agent distributed deadlock detection algorithm is not concerned with security, but instead leverages the fault tolerant advantages of mobile agents.

The benefits of this mobile agent solution include: asynchronous and autonomous operation, fault tolerance, network organization independence, protocol encapsulation, flexibility, locality of reference, inherent parallelism and scalability. Section 3.5 discusses the benefits provided by asynchronous and autonomous operation, fault tolerance, network independence and protocol encapsulation; therefore, they will not be explained in this section.

One of the additional benefits of this algorithm is flexibility. Due to the clear separation of policy (host environment) and mechanism (shadow and detector agents), upgrades can take place to the technique in part or in whole. Additionally, the platform independence of the solution makes it applicable to many possible targets.

The movement of the detector agents to the resources on which agents are blocked leverages locality of reference. This allows detector agents to perform local queries, extract relevant information and return to their point of origin without significant network use.

Each shadow agent involved in a distributed deadlock initiates the detection process when it detects the proper conditions. This property of the algorithm allows the global wait-for graph to be constructed in parallel. Each detector agent executes simultaneously to build the global wait-for graph, allowing each shadow agent to find where the deadlock cycle is and how to break the deadlock. An alternative to this technique is to serially send agents through the network gathering information.

The challenges associated with this technique consist of: detector agent movement, data movement, host environment standardization, lack of installed base, agent routing and speed. Detector agents are responsible for visiting locked resources and building portions of the global wait-for graphs. As a result, they have more movement than other agents in the system. If the size of the wait-for graph is large, this could require

significant movement of the detector agents. One possible solution to this problem is to increase the detector agents interval or to use gateway nodes.

In a similar manner to the movement challenge, if the size of the global wait-for graph is large, significant portions of it may be transported by the detector agents. The design goal of the algorithm is to keep the detector agents small and reactive to limit data movement through the network; therefore, the amount of data transported must be minimized.

The mobile agent hosts must coordinate the activities of the algorithm components; therefore, porting the solution also modifies the agent hosts. Due to the lack of standardization, this represents a significant amount of work and overhead when deploying a solution such as this. Additionally, the lack of an installed base of mobile agent hosts makes large scale deployment and testing difficult.

One of the fundamental aspects of this solution is the efficient routing of agents based on host environment and resources. If resources are allowed to move throughout the network, they must be tracked. Similarly, if dynamic topology updates are allowed, then the host-based routing tables must be updated. These two properties require that a complex routing protocol run in the background to ensure that agents are routed to the proper host, otherwise deadlock detection may fail.

Techniques such as this are implemented at a higher level than the raw socket layer or at the database level, as in some traditional solutions. This increased overhead and processing times can increase agent latency and lead to longer deadlock detection times. It should be noted that mobile agents are not normally deployed in systems where immediate deadlock detection is required.

3.6 Comparison of Proposed Solution to Traditional Solutions

The distributed deadlock detection technique presented in this thesis has unique properties that are adapted to the features of a mobile agent system. Additionally, some of the properties of the presented technique are similar to those found in traditional solutions. This section describes those properties and compares them to the related features of traditional solutions.

The presented mobile agent solution uses a trigger for "probe" initiation which is similar to the trigger used in edge-chasing solutions [20]. In both solutions, the detection of an entity that is waiting on an offsite resource is the probe trigger. Even though the initiation criteria are similar, the problems themselves and their mechanisms differ greatly. For example, traditional solutions have network organization (topology) assumptions to forward probes to the proper hosts, while the mobile agent solution makes no such assumption. Additionally, the traditional probe makes use of a protocol or message, while the presented mobile agent solution uses mobile detection agents.

Both traditional solutions and the presented technique do not centrally maintain a global wait-for graph [20]. Instead, the techniques maintain local wait-for graphs and build the global graph through a series of queries. Again, it is the form of these queries and the information they gather that differ between traditional and the presented mobile agent solution.

All techniques, whether they are traditional or mobile agent based, must have a deadlock resolution technique. Often this technique takes the form of transaction priorities or transaction numbering. These numbering schemes allow the deadlock detection solution to break the deadlock in a consistent manner.

Virtually all distributed systems and distributed deadlock detection techniques have lock coordination entities. For example, edge-chasing uses lock coordinators at each host [20] and the deadlock detection agent (DDA) technique by Krivokapic and Kemper [22] uses static agents to coordinate locks. In the presented mobile agent solution, the task of lock coordination is not static, but instead is mobile, in the form of shadow agents. Host environments simply coordinate the activities of agents based on the information presented by the shadows.

Traditional techniques and the presented solution require that a commitment protocol is executing to ensure that locks are not arbitrarily dropped during the deadlock detection process. Locks can only be dropped when agreement is reached between the concerned

parties. An example of this requirement is a two phase commit protocol running in the background.

3.7 Proposed Solution Theoretical Measurements

Assuming a fault free environment, it is possible to obtain theoretical values for the number of agent migrations and number of lock checks during the deadlock detection process. In-order or sequential execution must be maintained to predict these theoretical values. For the purposes of this discussion, consider a simple distributed deadlock situation. In this situation two agents (Agent 1, Agent 2), resources (Resource 1, Resource 2) and environments (environment 1, environment 2) are interacting to create a distributed deadlock.

Assuming that a deadlock condition exists and the assumed properties are true, the following steps must occur to detect the deadlock:

1. Detector 1 migrates to agent environment 2 and checks the locks for resource 1. Detector 2 migrates to agent environment 1 and checks its locked resource, resource 2.
2. Detector 1 learns that Agent 2 is blocked on resource 1 and returns to its parent shadow agent. Detector 2 learns that Agent 1 is blocked on resource 2 and returns to its parent shadow agent.
3. Shadow Agent 1 checks the returned information, determines that there is no deadlock and adds Agent 2 (and its info) to its deadlock table. Shadow Agent 2 checks the returned information, determines that there is no deadlock and adds Agent 1 (and its info) to its deadlock table.

4. Shadow Agent 1 and Shadow Agent 2 re-initialize their detectors and restart the process.
5. Detector 1 migrates to agent environment 2 and checks the locks for resource 1. Detector 2 migrates to agent environment 1 and checks its locked resource, resource 2.
6. Detector 1 learns that Agent 2 is blocked on resource 1 and Agent 1 is secondarily blocked on resource 1. Detector 2 learns that Agent 1 is blocked on resource 2 and Agent 2 is secondarily blocked on resource 2. Both detectors return to their parent shadow agents.
7. Shadow Agent 1 checks the returned information, notices that Agent 1 was returned in the list of blocked agents and flags the deadlock. Shadow Agent 2 checks the returned information, notices that Agent 2 was returned in the list of blocked agents and flags the deadlock.
8. Both Shadow Agents used a defined technique (i.e., priorities) to determine that resource 1 must be unlocked.
9. Shadow 1 initializes its detector to unlock resource 1 and to notify Agent 2 when the resource is unlocked.
10. Detector 1 migrates and unlocks the resource. Agent 2 is notified. Detector 1 returns to its parent and notifies it of unlock completion.

Before starting an analysis of the preceding sequence of events, several key concepts must be defined. A migration is defined as the movement of a mobile agent from one environment to another. Migrations are unidirectional; therefore, at least two migrations are required for an agent to check a lock and return.

A detector agent trip is defined as the number of messages required to visit all of the resources locked by a consumer agent and return. The number of migrations in a trip is

network topology dependent, but is always greater than the number of resources to visit. The act of migrating to a resource and returning is called a visit. In a similar manner to a trip, the number of migrations required to complete a visit is network topology dependent. The minimum number of migrations in a visit is two, but normally this value is higher.

The following variables can be defined to represent the various elements of the suggested distributed deadlock detection technique: Number of Agents in a Deadlock (N_A), Number of Repetitions (N_R), Detection Migration Volume (M_D), Resolution Migration Volume (M_R), Messages in a Trip (M_T) and Messages in a Visit (M_V).

Analysis of the steps required to complete the example deadlock detection case leads to the following general statements.

- $N_R = N_A$
- $M_D = N_A \times N_R \times M_T$
- $M_R = M_V$

In the example deadlock, N_A is 2 and M_T and M_V are 2. Applying these values gives: 2 repetitions, 8 migrations for detection and 2 migrations for resolution. By checking the event sequence, these values match the actual numbers found in the example.

Additional analysis shows that since N_R equals N_A the detection migration volume can be restated as: $N^2 \times (M_T)$, where N is the number of agents involved in the deadlock. It should be noted that these values are upper bounds for the volume of migrations and hence network load. They are not execution time estimates, since many factors not considered influence the time required to detect and resolve a deadlock using the suggested technique.

Due to the asynchronous nature of the distributed deadlock detection technique, the events suggested in the example do not execute in a sequential manner. Many of the migrations and processing required during the deadlock detection process occur in parallel and exploit locality of reference. This property means that estimating the total run time of the technique should not be based on the number of migrations, since many occur at the same time. Factors such as host processor speed, deadlock scanning implementation, network congestion and host environment efficiency have a significant impact on the time required to detect and resolve a deadlock.

Chapter 5 provides simulated measurements and discusses the impacts of properties such as topology, delays and faults on the theoretical message volumes. The impacts of changes in sequencing are significant, and move the number of detector agent migrations lower than the predicted theoretical value. Faults have the opposite effect, and push the number of detector agent migrations above the values predicted by the theoretical equations.

4.0 Mobile Agent Deadlock Modeling System

The distributed deadlock detection technique presented in chapter 3 was tested and validated in a mobile agent modeling system constructed for this thesis.

4.1 Simulator Platform Requirements

Many of the mobile agent system presented in section 2.5 were considered during the initial design of the modeling system. In particular, Aglets [4], Mozart [18], Mole [7] and an original implementation were considered. After analysis of the features and properties of the possible solutions, an original implementation was selected. Since the goal of this thesis is not widespread deployment of a solution, but instead a generic framework, this is an acceptable decision.

The following section briefly describes the technical selection criteria, and when applicable, how they apply to the considered solutions. The platform selection criteria were: resource use, movement between independent hosts, extendibility, traceability / instrumentation and platform independence. Each of the criteria and their respective weights are discussed below.

Resource use. The selected development platform must have low resource requirements to effectively simulate a large mesh of connected nodes. To validate the developed deadlock detection algorithm, boundary tests were performed. These tests required networks to be quickly constructed with configurations ranging from two to more than

twenty nodes. Resource requirements include CPU speed, memory and storage space. Additionally, the mobile agent hosts had to possibly execute on a single machine, making resource consumption a concern. Although none of the possible mobile agent systems had high resource requirements, an original implementation allowed complete control over resource consumption. For example, creating twenty Java virtual machines consumes significant resources and can not be easily hosted on a single machine.

Movement between independent hosts. One of the most significant requirements when simulating mobile agents is agent movement. Movement means that control of an agent is transferred between two independently executing mobile agent hosts. This requirement does not dictate that the hosts are physically separated, just that they do not coordinate or communicate except to transfer agent control. Additionally, a complex migration mechanism is not required, since it is purely an implementation detail and is different in many mobile agent systems. All of the considered mobile agent systems provided mechanism for agent migration across multiple machines or between virtual machines on a single machine.

Extendibility. Some of the properties of the developed deadlock detection algorithm require the host environment to perform coordination tasks related to shadow and consumer agents. As a result, the selected mobile agent system must allow extension of the agent host to ensure it can control the flow of agents. This could be provided by an explicit API or via hooks placed in the host code. Additional modifications are also required to support the routing scheme proposed by the solution. Modifying existing

solutions such as Aglets [4] or Mozart [18] is possible, but the amount of work did not vary significantly from creating an original implementation. Additionally, extending an existing solution requires detailed knowledge concerning the implementation of agent movement and control provided by that environment.

Traceability / Instrumentation. In order to measure the progress of the deadlock detection algorithm, it is required that the flow of agents be actively monitored and reported. The selected environment must provide an event model that can generate, gather and report logs from the various hosts that form the mobile agent network. Instrumentation points are required to gather performance related measurements. Again, this can be provided via an explicit API or via environment hooks, but an original implementation can be designed with measurability as a fundamental property. The same level of instrumentation can be achieved in existing systems, but additional software must be written to gather and report the information.

Platform independence. The goal of this thesis is to develop a mobile agent distributed deadlock detection technique that is applicable to many mobile agent systems. This goal is at risk by choosing any of the available mobile agent systems, since inevitably the properties and features of the selected environment are leveraged. Using functionality that is specific to a particular mobile agent system, language or host defeats this goal. An original implementation guarantees that the implementation does not rely on a particular feature or property of an existing system. Again, the same level of abstraction can be

achieved with an existing system, but additional software must be written to ensure the solution is separated from the specifics of a particular environment.

4.2 Simulator Implementation Overview

As was discussed in the previous section, an original implementation was selected to produce the simulation environment. To quickly implement the simulation environment an efficient development cycle (design, implementation and debugging) was essential. This requirement dictated that languages significantly different from C++ or Java are less desirable. Additionally, language features such as type safety, garbage collection and object orientation are important.

Languages such as Oz (Mozart) [18] were considered but rejected, due to their differences from traditional languages and steep learning curve. To meet the platform independence requirement, Java was selected as the programming language. This selection also meant that existing Java based mobile agent systems, such as Aglets [25] or Mole [30], could be easily adapted to implement the mobile agent solution.

To keep resource use to a minimum, it was decided that the implementation would use a single Java Virtual Machine and use threads to simulate independent agents, hosts and resources. This design decision facilitates agent mobility between hosts and agent tracking. Agent mobility can be viewed as the process of handing the control of a particular agent between two host threads. This property allows traditional solutions to be simulated and measured in the same environment as the mobile agent solution. It is a

simple extension to the design and implementation to serialize the agents and stream them to a remote mobile agent host via a socket.

To allow safe communication between the many threads that compose the simulator, inter-agent messaging had to be thread aware and thread safe. Blackboard based asynchronous messaging was selected to meet this requirement. Consumer, shadow and detector agents cannot communicate directly, they must post messages to the host environment's blackboard with all relevant information attached. This allows the thread safe (synchronized) sections of the simulator to be limited to the agent environment, which promotes easier design of the mobile agents themselves.

In the rare case that direct communication between threads is required, i.e., between an agent environment and a migrating mobile agent, the mobile agent thread is suspended before the direct communication is allowed. This ensures that the mobile agent's thread is in a safe state and all object locks (monitors) are freed before it is stopped. Once the communication is completed, the agents thread is restarted and execution begins exactly where it was stopped.

4.3 Simulator Architecture

The architecture of the simulation environment can be divided into four distinct areas or subsystems. These subsystems are: host environment, agent execution, resource management and simulator control. The following figure illustrates how these four subsystems interact to create and control a mobile agent.

Figure 5 Simulator Interactions

The basic premise of the simulation environment is that of asynchronous communication. Once the agent environments, routes and mobile agents have been created and configured, all elements run independently and asynchronously. This is ensured by creating each interacting element as an individual thread. Additionally, to simulate a large network such as the internet, random delays and errors are inserted before selected system events. This ensures that no two executions are identical and timing assumptions cannot be made.

All user interactions are controlled through a text based console. This console is responsible for receiving user input, interpreting it and sending it to the simulator for processing. The user interface components are also responsible for asynchronously gathering output from the mobile agents and environments to be displayed for the user.

Once the initial network has been configured and initialized, the user or script controls the actions of the agents by posting commands to the host blackboards. Since each agent and environment is an individual thread, the processing of the messages is done asynchronously from the action of posting them. This is not a general purpose implementation; therefore, the command set is limited to commands that are directly related to resource consumption. These commands include: "lock", "unlock", "move" and "die".

The host environment subsystem is responsible for coordinating resource consumption and the flow of agents. The host environments control resource consumption with the help of a dedicated resource manager. The resource manager is responsible for maintaining a list of resources and the owner of each resource. Note that the resource manager is not aware of agent location or the type of agent that is requesting a resource. If a locked resource is requested, the resource manager informs the host environment and specialized agent coordination activities to take place.

In addition to its resource management role, the host environment is responsible for controlling agent movement. This is done via defined method calls that are executed

each time an agent enters, wishes to move or leaves the host environment. Since these method calls are mandatory, the host environment can block agent movement or notify the agent of pending events upon arrival and exit.

Routing of agents is done via a dedicated topology manager which is programmed with available routes at setup time. Each time an agent notifies the host environment that it would like to move, it supplies the name of the target resource or host environment. The topology manager makes a best effort attempt to route the agent closer to its target. This routing is similar to that found in IP networks and will eventually route the agent to its target.

These three responsibilities of the host environment are combined to implement the control mechanism for the developed distributed deadlock detection technique. For example, consider the case when a consumer agent requests a lock on a resource. In this situation, the resource manager first checks to determine if the consumer agent has a shadow (via the token the consumer agent supplied upon arrival). If the consumer agent has a shadow, the host environment knows that it should also be present in the host. If the shadow is found, the request continues, if not, the request is denied.

Next, the resource manager checks the status of the requested resource. If the resource is unlocked, the consumer agent is passed the resource and the shadow agent is updated with the lock information. If the resource is already locked, the resource manager informs the host environment. The host environment in turn notifies the consumer agent

of the failure. At this point, the consumer agent may chose to block. If the consumer agent blocks, the host environment notifies the shadow agent to begin the deadlock detection process. If the consumer agent chooses not to block, but instead wishes to leave the host, it calls the "move" method. The topology manager picks the appropriate route and migrates the agent. The host environment migrates the consumer's shadow agent to ensure the agent is allowed to correctly operate in its new host.

As discussed in chapter three, the mobile agent simulator is populated with three types of mobile agents. Consumer, shadow and deadlock detection agents form the agent execution subsystem and share common properties. Each form of agent inherits from a common subclass the required host environment API methods. This ensures that all forms of mobile agents use host environment services to move through the network. Additionally, each agent inherits a blackboard message processing event loop. This event loop is responsible for removing messages sent to the agent from the blackboard and making them available for processing.

It is at the point of message processing that the different mobile agent subclasses are significantly different. Each agent is specialized to a specific task; therefore, they understand and process different directives. For example, consumer agents understand how to lock a resource and what to do when locks fail. Shadow agents are specialized to track the locks of a consumer agent and understand the messages sent to them by the host environment for this task. Finally, detection agents have the ability to visit a set of resources, gather resource lock information and return to their shadow agent.

Mobile agents, the host environment and resource manager must coordinate their activities to implement the distributed deadlock detection technique described in chapter three. After initial configuration and message sending, the user has no interaction with the simulator and only monitors the output for correct resolution. This simulates a set of autonomously operating agents in an environment such as the internet without worrying about the details of widespread deployment. The following chapters present traces and measurements which demonstrate the simulator's ability to detect and resolve distributed deadlocks.

5.0 Algorithm Measurements

This chapter presents measurements that were gathered using deadlock simulator presented in chapter 4. First, the simulator's measurements are presented and discussed. Next, a discussion of traditional solution measurements is presented. Finally, a discussion of the simulator measurements and how they should be interpreted is presented.

5.1 Mobile Agent Solution

Three significant properties of the mobile agent simulator were varied to obtain a range of measurements. These properties are: network organization, failure rate, and mobile system population. Varying these values provided a mechanism to ensure that the suggested solution works in any network organization, is not dependent on synchronous execution, is fault tolerant and can resolve difficult to find deadlocks.

Due to their significant differences and common deployment, mesh and ring network topologies were simulated and measured. Comparing the measurements between these configurations demonstrates that the suggested solution is network topology independent. Network organization independence is a significant advantage when the deployment of real-world mobile agent solutions is considered. Additional network organizations were used during validation, but are only variations on the ring and mesh configurations.

Measurements were gathered using the ring and mesh topologies in both fault-free and failure-prone configurations. The ability of the mobile agent solution to recover from failures demonstrates the desired property of survivability. Additionally, the measurements captured during failure-prone simulation allow the performance impact of faults to be determined.

Different combinations of agents and agent environments are used to measure the impact of the number of elements on the deadlock detection process. The configurations simulated and measured are presented in Table 9. Configurations 'a' through 'c' represent the most common numbers of elements which are involved in distributed deadlocks. Configuration 'd' represents a boundary test and demonstrates the ability of the mobile agent solution to detect and resolve deadlock involving a large number of elements.

Table 9 Simulation Environment Configurations

Configuration	Number of Agents	Number of Agent Environments
a	2	2
b	3	3
c	4	4
d	20	20

5.1.1 Measurement Methodology

To ensure that the measurement mechanism and simulation execution order do not have a significant influence on the end results, a consistent measurement methodology is

required. To satisfy this requirement, identical steps are used to create the deadlock situation in each simulation. While the deadlock steps remain constant, the properties introduced in the previous section are varied. This allows the impact of each property to be isolated and analyzed.

The simulation must model the dynamic environment found in real-world mobile agent networks in both fault-free and failure-prone configuration. The multi-threaded nature of the simulation engine ensures that synchronous execution will not occur, but the events can still execute in the same order over multiple executions. To prevent predictable ordering of the simulation, random delays are inserted at significant event points. Routing requests, mobile agent entrance, deadlock result reporting and lock creation were selected as delay checkpoints. Before each of these events is allowed to continue, the mobile agent requesting the event is forced to sleep for a random duration. Note that the maximum sleep value is configurable and is explained in the measurement section. Since the simulation environment is multi-threaded, these random delays do not prevent other threads from executing and produce a different sequence of events in each execution.

During the initial phase of each simulation, the desired number of agent environments are created and routes established based on the selected network organization. Next, a single agent and resource are added to each environment. All agents and resources begin as unlocked and available for consumption.

In the second phase of the simulation, each agent locks the resource found in its starting environment. Once the resource is locked, each agent migrates to the next available mobile agent host. For example, agent one starts in agent environment one and locks the local resource. Once this lock is complete, it moves to environment two. At this point the simulation is deadlock free, but all of the available resources are consumed.

In the third phase, each agent attempts to lock the resource in its new environment. Phase two ensures that these resources are not available and this set of lock attempts fail. The failure of a lock request moves an agent into its deadlock detection phase. Once the locks fail, each shadow agent begins the steps described in chapter three to detect and resolve the deadlock. Once the deadlock is detected and resolved phase three is completed.

Using all of the configurations listed in Table 9, these three phases are repeated for both the fault-free and failure-prone scenarios. The three phases and their steps were chosen to simulate a deadlock that requires the interaction and cooperation of all the agents in the system to detect and resolve the deadlock.

In each phase the shadow and detector agents keep statistics concerning the number of migrations, number of failures, number of completed resource checks and duration. Since random delays and free running threads are used, each measurement is repeated ten times and median values obtained to ensure that typical values are obtained.

5.1.2 Fault-Free Configuration Measurements

The measurements summarized in Table 10 and Table 11 were gathered using a fault-free configuration of the simulation environment. In this configuration agents may be delayed, but will not fail. To implement agent delays, each agent is forced to sleep for a random duration (up to 1500 milliseconds) at significant event points.

Each of the configurations listed in Table 9 are executed a total of ten times in a mesh configuration and ten times in a ring configuration. Total message volume and number repetitions are measured and median values for the ten executions are determined.

Total message volume represents the total number of migrations detector agents perform until the deadlock is detected and resolved. To check a single lock in a mesh organization, a detector agent must make two migrations. In a ring organization, more migrations are required to check a lock and return. The number of repetitions represents how many times an individual agent checked all of its locked resources and returned. Any number of migrations can make up a lock check repetition, since the number of migrations in a repetition is influenced by network topology and the number of resources an agent has locked.

One notable measurement that is not reported is duration. Due to the nature of the simulator and its random delays, this value has little relevance and does not map to real network values. Section 5.2 discusses how these measurements should be compared and interpreted.

Table 10 Mesh Connected and Fault Free Measurements

Configuration	Median Message Volume	Median Number of Repetitions
a	<ul style="list-style-type: none"> • 8 during detection • 2 during resolution 	<ul style="list-style-type: none"> • $A_1 : 2$ • $A_2 : 2$
b	<ul style="list-style-type: none"> • 18 during detection • 2 during resolution 	<ul style="list-style-type: none"> • $A_1 : 3$ • $A_2 : 3$ • $A_3 : 3$
c	<ul style="list-style-type: none"> • 32 during detection • 2 during resolution 	<ul style="list-style-type: none"> • $A_1 : 4$ • $A_2 : 4$ • $A_3 : 4$ • $A_4 : 4$
d	<ul style="list-style-type: none"> • 909 during detection • 2 during resolution 	<ul style="list-style-type: none"> • $A_{med} : 16$

Table 11 Ring Connected and Fault Free Measurements

Configuration	Median Message Volume	Median Number of Repetitions
a	<ul style="list-style-type: none"> • 8 during detection • 2 during resolution 	<ul style="list-style-type: none"> • $A_1 : 2$ • $A_2 : 2$
b	<ul style="list-style-type: none"> • 25 during detection • 4 during resolution 	<ul style="list-style-type: none"> • $A_1 : 3$ • $A_2 : 2$ • $A_3 : 3$
c	<ul style="list-style-type: none"> • 54 during detection • 5 during resolution 	<ul style="list-style-type: none"> • $A_1 : 4$ • $A_2 : 3$ • $A_3 : 3$ • $A_4 : 3$
d	<ul style="list-style-type: none"> • 5753 during detection • 21 during resolution 	<ul style="list-style-type: none"> • $A_{med} : 14$

5.1.3 Fault-Injected Measurements

The values in Tables 12 and 13 are measured with failures and faults enabled. In terms of the simulation, faults are represented by two types of events: long delays and detector agent failures. It is the responsibility of the simulator to insert random delays and failures at selected event points.

In a similar manner to the random delays in the fault-free configuration, the failure-prone simulation adds random delays to selected events. The difference in this configuration is the maximum delay value, which is four times longer than the fault-free case. A large maximum delay allows agents to get further out of sequence than they can with a shorter random delay. Longer delays model congested or faulty network links and ensure that the deadlock detection technique is not dependent on a particular execution order.

In addition to long delays, the failure-prone configuration can also cause complete detector agent failure. At selected events, the simulator randomly decides if a particular detector agent will be allowed to continue, or whether it should die. If the agent is killed, all of its information is lost and its parent shadow agent must detect the failure and create a new agent. The failure of a detector agent is a significant event, and represents the complete loss of the information gathered during a detection step. This is similar to dropping a probe in the traditional edge-chasing technique. Due to the fault tolerant design of the solution, the loss of a detector agent does not cause deadlocks to be missed and often does not incur significant extra work.

Each of the configurations listed in Table 9 is executed ten times in a mesh configuration and ten times in a ring configuration to gather the failure-prone measurements. Total message volume, number repetitions, total faults and phantom deadlocks are measured and median values determined.

Total message volume and number of repetitions are the same measurements gathered during the fault-free executions of the simulation. The additional measures of total faults and phantom deadlocks were added to the failure prone executions. The total faults represents the total number of detector agent failures that occurred during a simulation. Each detector agent failure has a direct impact on the message volume and number of repetitions, but do not cause the algorithm to lose information and miss the deadlock. The missed deadlocks measure indicates if any execution lost a deadlock due to a fault. Section 5.2 discusses how these measurements should be compared and interpreted.

Table 12 Mesh Connected and Failure Prone Measurements

Config #	Median Message Volume	Median Repetitions	Total Faults	Missed Deadlocks
a	<ul style="list-style-type: none"> • 8 during detection • 2 during resolution 	<ul style="list-style-type: none"> • $A_1 : 2$ • $A_2 : 2$ 	1	0
b	<ul style="list-style-type: none"> • 15 during detection • 2 during resolution 	<ul style="list-style-type: none"> • $A_1 : 3$ • $A_2 : 2$ • $A_3 : 2$ 	2	0
c	<ul style="list-style-type: none"> • 36 during detection • 2 during resolution 	<ul style="list-style-type: none"> • $A_1 : 2$ • $A_2 : 4$ • $A_3 : 5$ • $A_4 : 4$ 	11	0
d	<ul style="list-style-type: none"> • 1174 during detection • 2 during resolution 	<ul style="list-style-type: none"> • $A_{med} : 20$ 	58	0

Table 13 Ring Connected and Failure Prone Measurements

Config #	Median Message Volume	Median Repetitions	Total Faults	Missed Deadlocks
a	<ul style="list-style-type: none"> • 8 during detection • 2 during resolution 	<ul style="list-style-type: none"> • $A_1 : 2$ • $A_2 : 2$ 	1	0
b	<ul style="list-style-type: none"> • 33 during detection • 3 during resolution 	<ul style="list-style-type: none"> • $A_1 : 2$ • $A_2 : 4$ • $A_3 : 2$ 	13	0
c	<ul style="list-style-type: none"> • 66 during detection • 4 during resolution 	<ul style="list-style-type: none"> • $A_1 : 5$ • $A_2 : 3$ • $A_3 : 3$ • $A_4 : 3$ 	34	0
d	<ul style="list-style-type: none"> • 5939 during detection • 21 during resolution 	<ul style="list-style-type: none"> • $A_{med} : 16$ 	52	0

5.2 Measurement Discussion

In chapter three, theoretical values were derived for number of repetitions, number of migrations in detection and number of messages in resolution. In particular, the equations $N_R = N_A$, $M_D = N_A \times N_R \times M_T$ and $M_R = M_V$ were presented. These equations represent the maximum number of migrations and repetitions required in a synchronous, fault-free environment. Generally speaking, the data presented in this chapter supports the statement made in chapter three that failures push the number of migrations above the theoretical value and out-of-sequence processing pulls the number of migrations below the theoretical value.

For example, the value of M_D in a fault-free ring network of three agents (configuration b) should be: $M_D = 3 \times 3 \times 4$ or $M_D = 36$. The value presented in Table 10 shows a median value of 25 migrations in this configuration. The insertion of random delays into the simulation and their associated influence on event sequencing explains the difference between the measured and theoretical value for M_D . If an agent is sufficiently delayed, other agents can migrate and perform multiple lock checks before the delayed agent arrives. This allows the delayed agent to gather more information in a single trip than normally would be possible, speeding the detection of the deadlock. The effect of sequencing changes can also be seen in the number of repetitions (N_R), which are also below the theoretical maximum values.

The value presented for M_D in Table 13 for the failure-prone configuration of three agents in a deadlock is 33. Even with 13 total faults and long delays, this number is below the theoretical maximum of 36. This means that the additional work created by detection agent failures was offset by the savings achieved through event sequencing changes. As the number of faults increases, the number of migrations will approach and then exceed the theoretical maximum. The impact of faults on the total number of migrations is difficult to measure, since it depends on other factors such as network topology and at which point of the detection process the failure occurs.

The tables in section 5.1 highlight the impact of network topology on the total number of messages involved in distributed deadlock detection. For example, the number of migrations required to detect a deadlock in configuration 'c' (a ring topology) is 54 while the equivalent mesh topology requires 32 messages. Adding more nodes into this configuration, but keeping the number of agents constant, causes a proportional increase in both of these values. As a result the comparison of M_D between topologies is not a meaningful measurement. M_D should be compared between fault-free and failure-prone configurations to determine the impact of faults on the detection process.

The number of repetitions (N_R) is used to compare deadlock detection efficiency between topologies. N_R is only impacted by the number of agents in the system; therefore, if the number of agents is kept constant the number of required repetitions should stay constant between topologies. The values found in Table 10 and Table 11 for configuration 'c'

illustrate this property. Over the ten executions, agents in the mesh topology required 4 repetitions to detect the deadlock, while agents in the ring topology required about 3 repetitions. The difference in these values can be attributed to sequencing differences (and their savings); therefore, it can be determined that ring configurations are more sensitive to event sequence changes than mesh configurations. These values can be compared between fault-free and failure-prone scenarios to determine the impact of failures on the number of repetitions required to detect a distributed deadlock.

As was predicted in chapter three, the volume of migrations increases as a function of both the number of agents and the routes available in the topology. It should be noted that this increase in migration volume is spread between the nodes in the network and occurs in parallel. This distribution of additional migrations ensures that as the number of migrations increase, their impact on network congestion is limited.

Section 3.1 presents requirements and properties of a successful mobile agent distributed deadlock detection technique. These properties have a direct impact on the total number detector agent migrations required during distributed deadlock detection. In particular, network organization (topology) independence, fault tolerance and unrestricted agent movement increase the number of migrations. These properties provide flexibility and fault tolerance, but do not allow traditional technique assumptions, that can increase efficiency, to be made.

For example, if the suggested mobile agent algorithm assumes that network topology is static and always organized in a ring, a significant reduction in the number of migrations can be realized. Using this assumption, a simplistic deadlock detection technique can be created where a detector agent performs a walk of the entire network and gathers deadlock information. When the detector agent returns to its starting point, the data is checked for deadlocks. In this scheme, the number of migrations equals the number of nodes in the network. Unrestricted agent movement and fault tolerance cause similar increases in the number of migrations.

Most real world IP networks are similar to the mesh topology simulation; therefore, detector agents can often migrate directly to the desired host environment. As a result, the values in Table 12 are representative of actual deployment and are a compromise of flexibility and efficiency. Additionally, as suggested by [20] it has been observed and documented that most deadlocks occur between two transactions. Resource locked by a remote mobile agent roughly maps to a transaction in traditional distributed deadlock detection; therefore, the measurements for systems with two agents are of particular relevance. Due to the common nature of deadlocks between few agents or transactions, the number of messages needed to detect and resolve a deadlock between many agents is not a significant concern or disadvantage.

5.3 Traditional System Measurements

As a result of extensive research into distributed deadlock detection, many algorithms have been designed and implemented. One categorization of the many algorithms was

presented in section 2.3. Often distributed deadlock detection techniques are designed to solve a specific problem. As a result, their design focuses on a particular environment and the requirements of that system [22]. The product of this design philosophy is a deadlock detection algorithm that is specialized and tuned for a particular set of problems.

This specialization of distributed deadlock detection algorithms makes detailed comparison between algorithms difficult, unless they are designed for the same type of environment. The distributed deadlock detection technique developed in this thesis uses mechanisms and concepts similar to those found in traditional edge-pushing techniques; therefore, this section presents a high level comparison of the edge-pushing technique described in [20] and the proposed mobile agent solution.

Speed. Speed is a measurement that is only relevant to the system in which the distributed deadlock algorithm is executing. Some systems require deadlock detection in the order of milliseconds, while in others, resolution times measured in seconds are acceptable. Additionally, most implementations value correctness over raw speed and efficiency.

Message Volume. The number of messages required to detect and resolve a distributed deadlock using edge-pushing is $2(N - 1)$, where N is the number of transactions in the deadlock. The solution presented in this thesis requires $M_T (N^2)$ detector agent migrations to detect and resolve a deadlock, where N is the number of agents in the

deadlock. The additional messages are due to the lack of network organization assumptions that are inherent in the mobile agent solution. If an organization (such as a ring) was enforced by the mobile agent solution, the number of migrations is reduced to a value near the edge-pushing solution. The value of the flexibility and fault tolerance of the suggested solution outweigh the cost of the additional messages.

Fault Tolerance. Many traditional distributed deadlock algorithms do not consider failure recovery and do not guarantee correctness if faults are present [20]. As a result, the impact of faults during distributed deadlock detection is normally not measured. In the mobile agent solution presented in this thesis, the impact of faults is related to the number of migrations required to repeat a trip. This means that the impact of failures varies with network topology. In the two measured topologies, each failure caused less than a two percent increase in the number of migrations. Clearly the loss of efficiency is negligible and is counteracted by the benefits of fault recovery and the flexibility it provides.

6.0 Conclusions and Future Work

This chapter presents the conclusions drawn from the work completed for this thesis, followed by several suggestions for future work.

6.1 Conclusions

The problem stated in section 1.1 has been solved; as shown in sections 3.0 through 3.7, a technique has been developed to detect and resolve distributed deadlocks in mobile agent systems. If left unmodified, the assumptions of traditional distributed deadlock algorithms impose unacceptable restrictions on a mobile agent system or prevent a traditional solution from working successfully and / or reliably.

To successfully detect and resolve mobile agent distributed deadlocks, the solution suggested by this thesis leverages the advantages of the mobile agent paradigm. In particular, the significant properties of the technique that differentiate it from traditional solutions are: locality of reference, network organization independence, fault tolerance, asynchronous execution and freedom of movement. These attributes are realized through a platform independent mobile agent distributed deadlock detection solution in which the agents that consume resources are separated from the deadlock detection mechanism. This separation produces agents whose only tasks are deadlock initiation, detection and resolution. These agents are fully adapted to the properties of the mobile agent paradigm and collaborate to implement a complete distributed deadlock detection solution.

The second objective of this thesis is to define the properties and qualities of mobile agent solutions as compared to traditional distributed computing solutions. As discussed in sections 2.7 through 3.1, agent movement and fault frequency are the most significant influences on the development of a mobile agent solution. The systems in which mobile agent applications exist are dynamic, heterogeneous and unreliable; therefore, a mobile agent solution must have certain properties to execute effectively. These properties are: network organization independence, freedom of agent movement, asynchronous execution, survivability and limited coupling between solution and mobile agent system. The developed solution illustrates how these properties can be combined to produce a mobile agent solution to a traditional distributed computing problem.

Mobile agent environments demand network organization (topology) flexibility and fault tolerance. Incorporating these properties into a mobile agent solution impacts the amount of processing required for the algorithm. The measurements gathered during simulation of the solution suggested by this thesis confirm that these properties require additional processing and messaging. Although, due to the parallel nature of mobile agent systems, there is no conclusive evidence that the additional processing significantly impacts deadlock detection performance or efficiency. Additionally, the lack of similar mobile agent solutions make comparison and evaluation difficult and inconclusive.

6.2 Summary of Contributions

This thesis developed an original mobile agent solution to distributed deadlock detection. This solution represents the adaptation of the traditional distributed computing technique of edge–pushing on the global wait–for graph to the mobile agent paradigm.

Additionally, it demonstrates how the assumptions of traditional distributed techniques are impacted in mobile agent environments. These impacts are generalized and presented as challenges and properties of mobile agent solutions to distributed computing problems.

6.3 Future Work

There are several directions that future research could follow based on the work completed for this thesis. The solution presented in this thesis has three distinct phases: deadlock initiation, detection and resolution. Each of these phases communicates through well–defined information exchange. As a result, modifications to the phases can be individually investigated and implemented.

An alternative technique to the detector agent visits presented in this thesis is detector agent flooding. In this technique, detector agents still visit the resources locked by their shadow agent and determine which resources are held by any blocked agents. Instead of gathering information and returning, they clone themselves and send these clones to visit each of the resources held by blocked agents. This clone and visit phase would continue

until a detector agent clone finds the shadow agent that started the process. This phase would result in network flooding and would have to be carefully controlled to prevent network overloading. For certain topologies and configurations, this may represent a more efficient technique for detecting distributed deadlocks.

Changing solution variables, such as the shadow agent delay factor and detector agent periodic delays, allow the technique to be tuned to current network conditions. If these factors are continually monitored and adjusted, an adaptive, and possibly more efficient, solution can be developed.

This thesis provides theoretical and simulated measurements concerning the efficiency of the presented technique. Incorporating the solution into a widely deployed agents toolkit, such as Aglets [25], allows the duration impact and overhead of the suggested solution to be measured. Additionally, deploying the solution in a large network allows the impact of real-world network conditions to be measured.

The technique developed in this thesis makes no assumptions concerning the location of agents or data. This allows both agents and the resources they consume to move freely through the network without impacting the deadlock detection process. Further investigation is required to determine if resource movement is commonly required, and as a result, if the complexity of resource based routing is warranted.

Chapter 3 presents properties and qualities required for mobile agent solutions to traditional distributed computing problems. Although the requirements are presented in general terms, in-depth analysis was not completed for solutions other than distributed deadlock detection. The properties of additional mobile agent solutions to distributed computing problems, such as leader election or termination detection, should be analyzed and compared to those suggested by this thesis. Multiple solutions to similar problems would allow the technique to be generalized and patterns identified.

The solution presented in this thesis assumes that shadow agents are separate threads of execution from their target consumer agents. This separation ensures a generic solution, but dictates that the movement of shadow agents and consumer agents must be synchronized. If consumer agents are allowed to encapsulate multiple threads of execution, shadow agents can execute in tightly coupled thread, removing the synchronization requirements. Research into this area would couple the solution to a particular mobile agent system and implementation.

Finally, other deadlock resolution techniques, such as deadlock prevention, can be investigated to determine if they are also applicable to mobile agent environments. As the number of different deadlock detection techniques suggests, no solution is able to meet the requirements of all environments. Other techniques may be required to meet the needs of particular mobile agent environments and should be investigated.

Bibliography

- [1] Nwana, H. "Software Agents: An Overview." *Knowledge Engineering Review*, Vol. 11, No. 3, pp. 1–40, September 1996.
- [2] Franklin, S. and Graesser, A. "Is it an Agent, or just a Program?: A Taxonomy for Autonomous Agents." In *Proceedings of the Third International Workshop on Agent Theories, Architectures, and Languages*, Springer–Verlag, 1996.
- [3] Milojicic, D., Douglass, F. and Wheeler R. (ed.), *Mobility: Processes, Computers and Agents*. Addison–Wesley, Reading, 1999.
- [4] Lange, D. and Oshima, M., "Mobile Agents with Java: The Aglet API." *World Wide Web*, 1(3), September 1998, <http://www.baltzer.nl/www/contents/1998/1–3.html>.
- [5] Brustoloni, J. "Autonomous Agents: Characterization and Requirements." *Carnegie Mellon Technical Report CMU–CS–91–204*, Carnegie Mellon University, Pittsburgh, 1991.
- [6] White, J.E., "Telescript Technology: Mobile Agents." General Magic White Paper, Appeared in *Mobility: Processes, Computer and Agents*, Addison–Wesley, Reading, 1999.
- [7] Baumann, J., Holl, F., Rothermel, K. and Straber, M. "Mole–Concepts of a Mobile Agent Sytem." *World Wide Web*, 1(3), September 1998, <http://www.baltzer.nl/www/contents/1998/1–3.html>.
- [8] Milojicic, D.S., Chauhan, D. and laForge, W. "Mobile Objects and Agents (MOA), Design Implementation and Lessons Learned." *Proceedings of the 4th USENIX Conference on Object–Oriented Technologies (COOTS)*, pp. 179–194, April 1998.
- [9] OMG. "The OMG MASIF Standard." *World Wide Web*, September 1998, <http://www.fokus.gmd.de/research/cc/ecco/masif/>
- [10] Hohl, F. "The Mobile Agent List", *World Wide Web*, November 1999, <http://mole.informatik.uni-stuttgart.de/mal/mal.html>
- [11] Cabri, G., Leonardi, L. and Zambonelli, F. "Mobile–Agent Coordination Models for Internet Applications." *IEEE Computer*, Vol. 33, No. 2, pp. 82–89, February 2000.
- [12] Joshi, N. and Ramesh, VC. "On Mobile Agent Architectures." *World Wide Web*, October 1998, <http://vcr.iit.edu/papers/mobileagents/mobileagents.html>
- [13] Dale, J. "A Mobile Agent Architecture for Distributed Information Management." *World Wide Web*, September 1997, <http://www.mmrg.ecs.soton.ac.uk/publications/archive/dale1997a/html/papers/thesis.html>

- [14] Lange, D. and Oshima, M. "Seven Good Reasons for Mobile Agents." *Communications of the ACM*, Vol. 42, No. 3, pp. 88–89, March 1999.
- [15] Ciancarini, P. and Vitali, R. "A case study in designing a document–centric groupware application over the Internet." *Proceedings of The Active Web*, January 1999, <http://www.visualize.uk.com/conf/activeweb/proceed/contents.asp>.
- [16] Johansen, D., Renesse, R. and Schneider, F. "Operating System Support for Mobile Agents.", Appeared in *Mobility: Processes, Computer and Agents*, Addison–Wesley, Reading, 1999.
- [17] Murphy, A. "Algorithm Development in the Mobile Environment." *World Wide Web*, 1999, <http://swarm.cs.wustl.edu/~alm/papers/icse99DS/index.html>.
- [18] DFKI, Universität des Saarlandes, Swedish Institute of Computer Science and Université catholique de Louvain. "The Mozart Programming System." *World Wide Web*, February 2000, <http://www.mozart-oz.org>.
- [19] Franklin, S. "Coordination without Communication." Institute for Intelligent Systems and Department of Mathematical Sciences University of Memphis, *World Wide Web*, August 1999, <http://www.msci.memphis.edu/~franklin/coord.html>.
- [20] Coulouris, G., Dollimore, J. and Kindberg, T. *Distributed Systems Concepts and Designs*. 2nd ed. Addison–Wesley, Don Mills, Ontario, 1994.
- [21] Peine, H. and Stolpmann, T. "The Architecture of the Ara Platform for Mobile Agents." Appeared in *Mobility: Processes, Computer and Agents*. Addison–Wesley, Reading, 1999.
- [22] Krivokapic, N. and Kemper, A. "Deadlock Detection Agents: A Distributed Deadlock Detection Scheme." Technical Report MIP–9617, Universität Passau, 1996.
- [23] Tanenbaum, A. *Modern Operating Systems*. Prentice Hall Inc., Englewood Cliffs, 1992.
- [24] Holliday, J. and El Abbadi, A. "Distributed Deadlock Detection." University of California at Santa Barbara, *World Wide Web*, September 2000, http://www.cs.ucsb.edu/~joanne46/dd_9_16.htm.
- [25] IBM Research. "Aglets Software Development Kit." *World Wide Web*, August 2000, <http://www.trl.ibm.co.jp/aglets/>
- [26] Kotz, D., Gray, R., Nog, S., Rus, D., Chawla, S. and Cybenko, G. "Agent TCL: Targeting the Needs of Mobile Computers.", Appeared in *Mobility: Processes, Computer and Agents*, Addison–Wesley, Reading, 1999.
- [27] Mobile Agents at Dartmouth College. "D'Agents Dartmouth." *World Wide Web*, May 2000, <http://agent.cs.dartmouth.edu/>

- [28] Walsh, T., Paciorek, N. and Wong, D. "Security and Reliability in Concordia.", Appeared in *Mobility: Processes, Computer and Agents*, Addison-Wesley, Reading, 1999.
- [29] Mitsubishi Electric Information Technology Center. "Concordia – Java Mobile Agent Technology." *World Wide Web*, January 2000, <http://www.meitca.com/HSL/Projects/Concordia/>
- [30] University of Stuttgart. "The Home of the Mole." *World Wide Web*, September 2000, <http://mole.informatik.uni-stuttgart.de/>
- [31] University of Tromsø and Cornell University. "TACOMA – Operating System Support For Mobile Agents." *World Wide Web*. August 1999. <http://www.tacoma.cs.uit.no/>
- [32] ObjectSpace Inc. "ObjectSpace Voyager Core Package Technical Overview.", Appeared in *Mobility: Processes, Computer and Agents*, Addison-Wesley, Reading, 1999.
- [33] ObjectSpace Inc. "ObjectSpace Product Information: Voyager." *World Wide Web*. September 2000. <http://www.objectspace.com/products/voyager/>
- [34] Fachbereich Informatik and Johann-Wolfgang-Goethe-Universitaet Frankfurt. "ffMain". *World Wide Web*. February 2000. <http://www.tm.informatik.uni-frankfurt.de/Projekte/MA/>
- [35] University of Geneva. "The Messenger Proejct." *World Wide Web*. December 1997. <http://cui.unige.ch/tios/msgsr/>
- [36] General Magic, Inc. "Odyssey". *World Wide Web*. November 2000. <http://www.genmagic.com/>
- [37] University of Modena. "MARS (Mobile Agent Reactive Space)." *World Wide Web*. October 2000. <http://sirio.dsi.unimo.it/MOON/MARS/index.html>
- [38] Cardelli, L. "Mobile Computational Ambients." *World Wide Web*. September 2000. <http://www.luca.demon.co.uk/Ambit/Ambit.html>
- [39] TU Berlin and University of Bologna. "The PageSpace Effort." *World Wide Web*. January 1997. <http://flp.cs.tu-berlin.de/pagespc/>
- [40] Flanagan, D. *Java in a Nutshell*. O'Reilly & Associates, Inc., Cambridge, 1996.
- [41] Horstmann, C. and Cornell, G. *Core Java Volume I – Fundamentals*. Sun Microsystems Press, California, 1999.
- [42] Sun Microsystems Inc. "Java 2 Platform v1.2.2 API Specification." *World Wide Web*. September 1999. <http://java.sun.com/products/jdk/1.2/docs/api/index.html>

- [43] Aridor, Y. and Oshima, M. "Infrastructure for Mobile Agents: Requirements and Design." Springer-Verlag. *Proceeding of the 2th International Workshop on Mobile Agents*, September 1998.
- [44] Tel, G. and Friedemann, M. "The Derivation of Distributed Termination Detection Algorithms from Garbage Collection Schemes." *ACM Transactions on Programming Languages and Systems*, Vol. 15, No. 1, pp. 1–35, January 1993.

Simulator Execution Trace

The following output is a simulation trace for a mesh topology populated by three agents and three resources. Deadlock initialization, detection and resolution are captured. This trace is indicative of the execution used to gather the measurements presented in chapter five.

```
(Sat Oct 28) Resource r2.db locked by Agent2
(Sat Oct 28) Resource r3.db locked by Agent3
(Sat Oct 28) env1 migrated agent "Agent1" to environment: env2
(Sat Oct 28) env1 migrated agent "Agent1 shadow" to environment: env2
(Sat Oct 28) env2 migrated agent "Agent2" to environment: env3
(Sat Oct 28) env2 migrated agent "Agent2 shadow" to environment: env3
(Sat Oct 28) env3 migrated agent "Agent3" to environment: env1
(Sat Oct 28 ) env3 migrated agent "Agent3 shadow" to environment: env1
(Sat Oct 28) ---> Agent: Agent3 blocked on resource: r1.db
(Sat Oct 28) ---> Agent: Agent2 blocked on resource: r3.db
(Sat Oct 28) ---> Agent: Agent1 blocked on resource: r2.db
(Sat Oct 28) ----- Agent2 shadow starting detection phase
-----
(Sat Oct 28) ----- Agent1 shadow starting detection phase
-----
(Sat Oct 28) ----- Agent3 shadow starting detection phase
-----
(Sat Oct 28) Agent: Detector for Agent2 shadow Starting Deadlock Detection
Process
(Sat Oct 28) Agent: Detector for Agent3 shadow Starting Deadlock Detection
Process
(Sat Oct 28) Agent: Detector for Agent1 shadow Starting Deadlock Detection
Process
(Sat Oct 28) env3 migrated agent "Detector for Agent2 shadow" to environment:
env2
(Sat Oct 28) env1 migrated agent "Detector for Agent3 shadow" to environment:
env3
(Sat Oct 28) env2 migrated agent "Detector for Agent1 shadow" to environment:
env1
(Sat Oct 28) Agent: Detector for Agent3 shadow all resources checked.
Returning.
(Sat Oct 28) env3 migrated agent "Detector for Agent3 shadow" to environment:
env1
(Sat Oct 28) Agent: Detector for Agent2 shadow all resources checked.
Returning.
```



```

(Sat Oct 28) env2 migrated agent "Detector for Agent2 shadow" to environment:
env3
(Sat Oct 28) Agent3 shadow reported ** No Deadlock **
(Sat Oct 28) ----- Agent3 shadow re-starting detection phase
-----
(Sat Oct 28) Agent: Detector for Agent3 shadow Starting Deadlock Detection
Process
(Sat Oct 28) Agent: Detector for Agent1 shadow all resources checked.
Returning.
(Sat Oct 28) env1 migrated agent "Detector for Agent1 shadow" to environment:
env2
(Sat Oct 28) Agent2 shadow reported ** No Deadlock **
(Sat Oct 28) ----- Agent2 shadow re-starting detection phase
-----
(Sat Oct 28) %%% Agent: Detector for Agent3 shadow died %%%
(Sat Oct 28)
Stats for Agent "Detector for Agent3 shadow":
    Msgs in detection: 3
    Msgs in resolution: 0
(Sat Oct 28) env1 migrated agent "Detector for Agent3 shadow" to environment:
env3
(Sat Oct 28) Agent: Detector for Agent2 shadow Starting Deadlock Detection
Process
(Sat Oct 28) Agent1 shadow reported ** No Deadlock **
(Sat Oct 28) ----- Agent1 shadow re-starting detection phase
-----
(Sat Oct 28) Agent: Detector for Agent1 shadow Starting Deadlock Detection
Process
(Sat Oct 28) env3 migrated agent "Detector for Agent2 shadow" to environment:
env2
(Sat Oct 28) env2 migrated agent "Detector for Agent1 shadow" to environment:
env1
(Sat Oct 28) Agent: Detector for Agent1 shadow all resources checked.
Returning.
(Sat Oct 28) Agent: Detector for Agent2 shadow all resources checked.
Returning.
(Sat Oct 28) env1 migrated agent "Detector for Agent1 shadow" to environment:
env2
(Sat Oct 28) env2 migrated agent "Detector for Agent2 shadow" to environment:
env3
(Sat Oct 28) Agent1 shadow reported ** No Deadlock **
(Sat Oct 28) ----- Agent1 shadow re-starting detection phase
-----
(Sat Oct 28) Agent: Detector for Agent1 shadow Starting Deadlock Detection
Process
(Sat Oct 28) Agent2 shadow reported ----> Deadlock <----
(Sat Oct 28) Deadlock Cycle: r1.db -> r2.db -> r3.db ->
(Sat Oct 28 12:53:26 EDT 2000) r1.db at env1 with priority 0 must be unlocked
and Agent3 should be notified

```

```

(Sat Oct 28) env2 migrated agent "Detector for Agent1 shadow" to environment:
env1
(Sat Oct 28) Agent: Detector for Agent1 shadow all resources checked.
Returning.
(Sat Oct 28) env1 migrated agent "Detector for Agent1 shadow" to environment:
env2
(Sat Oct 28) #### "Agent3 shadow" detected  dead detector .. restarting ####
(Sat Oct 28) Agent: Detector for Agent3 shadow Starting Deadlock Detection
Process
(Sat Oct 28) Agent1 shadow reported ** No Deadlock **
(Sat Oct 28 ) ----- Agent1 shadow re-starting detection phase
-----

(Sat Oct 28 ) Agent: Detector for Agent1 shadow Starting Deadlock Detection
Process
(Sat Oct 28) env1 migrated agent "Detector for Agent3 shadow" to environment:
env3
(Sat Oct 28 ) env2 migrated agent "Detector for Agent1 shadow" to environment:
env1
(Sat Oct 28) Agent: Detector for Agent3 shadow all resources checked.
Returning.
(Sat Oct 28 ) env3 migrated agent "Detector for Agent3 shadow" to environment:
env1
(Sat Oct 28) Agent: Detector for Agent1 shadow all resources checked.
Returning.
(Sat Oct 28) env1 migrated agent "Detector for Agent1 shadow" to environment:
env2
(Sat Oct 28) Agent3 shadow reported ----> Deadlock <----
(Sat Oct 28) Deadlock Cycle: r2.db -> r3.db -> r1.db ->
(Sat Oct 28) r1.db at env1 with priority 0 must be unlocked and Agent3 should
be notified
(Sat Oct 28) Agent1 shadow reported ** No Deadlock **
(Sat Oct 28) ----- Agent1 shadow re-starting detection phase
-----

(Sat Oct 28) Agent: Detector for Agent1 shadow Starting Deadlock Detection
Process
(Sat Oct 28) env2 migrated agent "Detector for Agent1 shadow" to environment:
env1
(Sat Oct 28) Agent: Detector for Agent1 shadow all resources checked.
Returning.
(Sat Oct 28) env1 migrated agent "Detector for Agent1 shadow" to environment:
env2
(Sat Oct 28) Agent1 shadow reported ----> Deadlock <----
(Sat Oct 28) Deadlock Cycle: r3.db -> r1.db -> r2.db ->
(Sat Oct 28) r1.db at env1 with priority 0 must be unlocked and Agent3 should
be notified
(Sat Oct 28) **** Agent1 shadow must break  the deadlock ****
(Sat Oct 28) ===== Agent1 shadow starting resolution phase
=====

(Sat Oct 28) env2 migrated agent "Detector for Agent1 shadow" to environment:
env1

```

(Sat Oct 28) ===== Detector for Agent1 shadow resolution phase ended, resource rl.db is unlocked =====

(Sat Oct 28) Resource rl.db unlocked by Agent1

(Sat Oct 28) Resource rl.db locked by Agent3

(Sat Oct 28) env1 migrated agent "Detector for Agent1 shadow" to environment: env2

(Sat Oct 28) Agent: Detector for Agent1 shadow Notified Parent of Unlock Completion

(Sat Oct 28)

Stats for Agent "Agent1 shadow":

Number of Lock Checks: 5

Number of Fault Recoveries: 0

(Sat Oct 28)

Stats for Agent "Detector for Agent3 shadow":

Msgs in detection: 2

Msgs in resolution: 0

(Sat Oct 28)

Stats for Agent "Detector for Agent1 shadow":

Msgs in detection: 10

Msgs in resolution: 2

(Sat Oct 28)

Stats for Agent "Agent3 shadow":

Number of Lock Checks: 2

Number of Fault Recoveries: 1

(Sat Oct 28)

Stats for Agent "Agent2 shadow":

Number of Lock Checks: 2

Number of Fault Recoveries: 0

(Sat Oct 28)

Stats for Agent "Detector for Agent2 shadow":

Msgs in detection: 4

Msgs in resolution: 0