Distributed Deadlock Detection in Mobile Agent Systems

Bruce Ashfield, Dwight Deugo, Franz Oppacher, Tony White

Carleton University, School of Computer Science,
Ottawa, Ontario, Canada, K1S 5B6
Bruce_Ashfield@Mitel.COM, {deugo,oppacher,white}@scs.carleton.ca

Abstract: Mobile agent systems have unique properties and characteristics and represent a new application development paradigm. Existing solutions to distributed computing problems, such as deadlock avoidance algorithms, are not suited to environments where both clients and servers move freely through the network. This paper describes a distributed deadlock solution for use in mobile agent systems. The properties of this solution are locality of reference, topology independence, fault tolerance, asynchronous operation and freedom of movement. The presented technique, called the shadow agent solution proposes dedicated agents for deadlock initiation, detection and resolution. These agents are fully adapted to the properties of a mobile agent environment.

1 Introduction

The increasing use of mobile agents for application development often stresses or breaks traditional distributed computing techniques [1]. Many existing solutions to problems [2], such as deadlock avoidance or leader election, are not suited for environments where both clients and servers move freely throughout the network. The mobility of these entities creates interactions and new conditions for distributed algorithms and applications that are traditionally not issues for them. The fundamental building blocks of many traditional distributed algorithms rely on assumptions, such as data location, message passing and static network topology. The movement of both clients and servers in mobile agent systems breaks these fundamental assumptions. Simply forcing traditional solutions into mobile agent systems changes the dynamic nature of their environments by imposing restrictions, such as limiting agent movement. As a result, new effective and efficient techniques for solving distributed problems are required that target mobile agent systems.

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) [3], extending and enhancing the data-service relationship by adding movement to both services and data. Generally speaking, mobile agents, such as those involving consensus, data transfer and distributed database processing must coordinate with one another to provide services and access data. The advanced synchronization required in these mobile agent based applications can lead to complex, multi-node deadlock conditions that must be detected and resolved. Traditional distributed deadlock schemes [4] fail when agent mobility and

faults are added to deadlock resolution requirements. Moreover, due to their assumptions, traditional techniques such as edge chasing on the global wait-for graph, are inadequate solutions in a mobile agent system.

In this paper, we develop a solution to the traditional problem of deadlock detection and resolution targeted for mobile agent systems. The presented solution is adapted from wait-for graph algorithms [5] to work in mobile agent environments requiring fault tolerance, asynchronous execution and absence of assumptions concerning the location or availability of data. Our solution is not bound to a particular mobile agent environment or toolkit. Therefore, properties specific to a particular mobile agent system do not influence the solution. Efficiency and speed are secondary concerns in this model, but are considered and discussed when applicable.

Section two provides a general overview of the problem of distributed deadlocks in mobile agent systems. Section three presents our approach to distributed deadlock detection in a mobile agent environment. Section four discusses the simulation experiments that were done to validate the proposed solution. Finally, in section five we draw our conclusions.

2 Deadlock

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" [4, 5]. 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 continue operating.

There are four conditions that are required for a deadlock [4]:

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 [4]. Ignoring deadlocks is the easiest scheme to implement. Deadlock detection attempts to locate and resolve deadlocks. 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 prevention is the structuring of a system in such a manner that one of the four necessary conditions for deadlock cannot occur [5]. Each solution category is suited to a specific type of environment and has advantages and disadvantages. In this paper, we focus on

deadlock detection and resolution, which is the most commonly implemented deadlock solution.

2.1 Mobile Agent issues with Traditional Deadlock Detection

Agent movement and failure are two properties of a mobile agent system that traditional distributed deadlock detection approaches don't consider. For example, resources and their consumers in traditional distributed deadlock detection do not move through the network and each server has knowledge about the location of other 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 [6]. The mobile agent and the host server can continue to interact with additional network resources [7] 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 [8], 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. 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, as are 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.

Often, distributed problem solving techniques have not dealt well with failures or crashes. For example, the distributed deadlock detection algorithm described by [5] states "The above algorithm should find any deadlock that occurs, provided that waiting transactions do no abort and there are no failures such as lost messages or servers crashing." The nature of mobile agents and their operating environment is commonly failure prone. Therefore, algorithms with strict fault assumptions will have difficulty functioning in many mobile agent systems.

3 Deadlock Shadow Agent Approach

The following sections describe our approach to distributed deadlock detection in mobile agent environments. We assume the following:

- 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 the nodes are connected.
- The topology of the network is static once the algorithm begins.
- We assume 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.

- 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 consumer agent's resource requests to its associated deadlock detection counterparts.
- 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 and many readers problem).
- 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 authority does not automatically block the agent, since this would limit flexibility and limit the dynamic nature of the mobile agent environment.
- 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 the host environment approves their request.
- 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).

3.1 Algorithm Overview

Our mobile agent algorithm uses agent-adapted mechanisms that are based on those found in traditional edge-pushing global wait-for graph algorithms. 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 Agent: The only agent in the system that actively performs tasks and consumes or locks resources. It represents the entity that implements algorithms. It has no active role in deadlock detection and resolution.
- Shadow Agent: This agent is created by host environments and is responsible for maintaining the resources locked by a particular consumer agent, following it through the network and for initiating the deadlock detection phase. In addition, it analyses the data gathered by detection agents to initiate deadlock resolution and detects and recovers from faults during

the deadlock detection process. It represents a portion of the global wait-for graph.

- Detection Agent: Shadow agents create this agent when informed by the host environment that their target agent has blocked. They are small, lightweight mobile agents that are responsible for visiting hosts and building the global wait-for graph and for breaking the deadlock situation.

The Host Environment is responsible for hosting the mobile agents and provides APIs to access, move, block, and unblock resources. It 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.

In our approach, shadow agents maintain a part of the wait-for graph on behalf of their target consumer agent. During the deadlock detection process shadow agents continually check for deadlocks in the global wait-for graph and create deadlock detection agents to build the global graph.

3.2 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 move freely 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 agent requests an exclusive resource lock. 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, .. , R_n] -> R_n. 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, .. , R_n] are a form of distributed transaction and

represent the resources that are held exclusively by a particular consumer agent. Finally, the R_b notation following the arrow (->) represents the resource on which an agent is blocked. If a consumer agent is not blocked, the -> R_b portion of the notation is not present. Combining all of the local transactions (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. A consumer agent cannot perform the following actions if its associated shadow agent is not present: move, lock, unlock. The consumer is notified of the failure and the request must be resubmitted. This restriction ensures that the shadow agents will contain the exact state of the wait-for graph, even if they are delayed during transmission.

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.

The host environment informs shadow agents that their target has block or unblocked through a defined 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 shadow agent initiates the distributed deadlock detection sequence, otherwise, no special processing occurs.

3.3 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, summarized as follows:

- **Agent Name:** The unique identifier of the consumer agent that this information belongs to
- **Blocked Resource:** The resource that the agent is blocked on, which includes the resource's unique name, the consumer agent that has this resource locked, the environment's name that contains this resource, and the priority of this resource.
- **Primary Locks:** The list of primary locks (resources) held by this agent.

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 analyses 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 initialised 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 information summaries maintained by shadow agents minimize the impact of failures. 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.

If a shadow agent finds a deadlock in the global wait-for graph, the deadlock resolution sequence is initiated.

3.4 Deadlock Resolution

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.

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 1.
2. The primary locks held in the root detection table entry form the next nodes in the graph. This is account.db in Figure 1.
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 1.
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 1 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 1 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.

Agent	Blocked Resource	Primary Locks
A1	account.db	[user.db]
A2	user.db	[test.db, payroll.db]
A3	test.db	[account.db]
A6	accound.db	[foo.db]

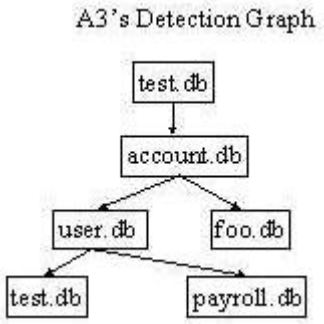

Fig. 1. Deadlock Detection Information

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. An example of this technique is when each node is added to the mobile agent system 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. Another option is, 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 owners 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 host environment carries out the final step of the deadlock resolution. When the host environment notices that a resource that 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.

4 Experiments

We tested and validated the distributed deadlock detection approach using a mobile agent modelling system simulator we developed. The simulator had four properties that could be changed: network topology, failure rate, population and the type of deadlock. Varying these values provided a mechanism to ensure that the suggested approach works with any topology, is not dependent on synchronous execution, is fault tolerant and can resolve difficult to find deadlocks.

Mesh and ring network topologies were selected due to their significant differences and common deployment. Comparing the measurements between these configurations demonstrated that the suggested solution is topology independent. This property is significant when the context of mobile agent applications is considered. Additional topologies 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 configuration. They indicated the survivability of the approach and measured the performance impact that faults have on the deadlock detection process. In addition, four configurations of agents and agent environments were used to measure the impact of increasing the number of elements in the deadlock detection process. It should be noted that the same type of deadlock situation was measured while varying the topology and fault properties of the simulator.

Most real world IP networks are similar to the mesh configuration simulation, meaning a detector agent can often migrate directly to a desired host environment. Moreover, most deadlocks occur between two transactions [5]. A locked resource by a remote agent roughly maps to a transaction in traditional distributed deadlock detection. Therefore, the measurements for systems with two agents are of particular relevance. The complete set of experiments and results are found in [9].

5 Conclusions

The assumptions of traditional distributed deadlock algorithms prevent them from completing successfully in a mobile agent environment. To successfully detect and resolve mobile agent distributed deadlocks, our approach leverages the advantages of the mobile agent paradigm. In particular, the principle properties of the developed technique that differentiate it from traditional solutions are: locality of reference, topology independence, fault tolerance, asynchronous operation and freedom of movement. These attributes are realized through a platform independent, mobile agent distributed deadlock detection solution. The agents that consume resources in the mobile agent system are separated from the deadlock detection mechanism. This

separation produces dedicated agents for deadlock initiation, detection and resolution. These agents are fully adapted to the properties of the mobile agent environment and collaborate to implement a complete distributed deadlock detection solution.

Mobile agent environments demand topology flexibility and fault tolerance. Incorporating these properties into mobile agent solutions impacts performance. The measurements gathered during simulations of our technique confirm that these properties require additional processing and messages. Due to the parallel nature of mobile agent environments, there is no conclusive evidence that these additional messages significantly impact deadlock detection performance or efficiency. Additionally, the lack of similar mobile agent solutions makes comparison and evaluation difficult and inconclusive.

References

1. Tel, G.: Introduction to Distributed Algorithms, Cambridge University Press (1994)
2. Lynch, N.: Distributed Algorithms, Morgan Kaufmann Publishers Inc, San Francisco, California, (1996)
3. Silva, A., Romão, A., Deugo, D., and Da Silva, M.: Towards a Reference Model for Surveying Mobile Agent Systems. Autonomous Agents and Multi-Agent Systems. Kluwer Online, (2000)
4. Tanenbaum, A: Modern Operating Systems. Prentice Hall Inc., Englewood Cliffs, (1992)
5. Coulouris, G., Dollimore, J. and Kindberg, T.: Distributed Systems Concepts and Designs. 2nd ed. Addison-Wesley, Don Mills, Ontario, (1994)
6. 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), (1998), 179-194
7. White, J.E.: Telescript Technology: Mobile Agents, Mobility: Processes, Computer and Agents, Addison-Wesley, Reading, (1999)
8. Krivokapic, N. and Kemper, A.: Deadlock Detection Agents: A Distributed Deadlock Detection Scheme, Technical Report MIP-9617, Universität Passau, (1996)
9. Ashfield, B.: Distributed Deadlock Detection in Mobile Agent Systems, M.C.S. Thesis, Carleton University, (2000)