A Distributed Deadlock Detection and Resolution Algorithm Using Agents

by

Mani Samani

B.Sc., University of Isfahan, 2011

THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
IN
COMPUTER SCIENCE

UNIVERSITY OF NORTHERN BRITISH COLUMBIA

December 2017

Abstract

Deadlock is an intrinsic bottleneck in Distributed Real-Time Database Systems (DRTDBS). Deadlock detection and resolution algorithms are important because in DRTDBS, deadlocked transactions are prone to missing deadlines. We propose an Agent Deadlock Detection and Resolution algorithm (`ADCombine`), a novel framework for distributed deadlock handling using stationary agents, to address the high overhead suffered by current agent-based algorithms. We test a combined deadlock detection and resolution algorithm that enables the Multi Agent System to adjust its execution based on the changing system load, and that selects its victim transactions more judiciously. We demonstrate the advantages of `ADCombine` over existing algorithms that use agents or traditional edge-chasing through simulation experiments that measure overhead and performance under a widely varying of experimental conditions.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

ACRONYMS

2PC Two-Phase Commit. vi, 17, 18, 71, 123, 126, 127

2PL Two-Phase Locking. vi, 15, 16

ACID Atomicity, Consistency, Isolation, and Durability. 2, 7, 13

ADCombine Agent Deadlock Combined Detection and Resolution Algorithm. 9, 54, 57, 95, 96, 97, 99, 100, 101, 103, 104, 105, 130

ADDetect Agent Deadlock Detection Algorithm. vi, 8, 40, 41, 43, 47, 49, 57, 85, 87, 88, 89, 90, 91, 94, 95, 104, 105

ADRes Agent Deadlock Resolution Algorithm. vi, 8, 47, 49, 50, 54, 64, 95, 100, 104

BDI Belief-Desire-Intention. 24, 29

BIAMAP Bidirectionally Initiated Action MAP. 27, 28

CCP Concurrency Control Protocol. 14, 60, 61, 67, 69

CNP Contract Net Protocol. 27, 29

D2PL Dynamic Two-Phase Locking. 16, 69, 72

DAI Distributed Artificial Intelligence. 29

DCOP Distributed Constraint Optimization Problem. 25

DDBS Distributed Database System. 1, 10, 11, 17, 18, 30, 38, 41, 43, 50, 58, 61, 70, 77, 82, 103

DEC-POMDP Decentralized Partially Observable Markov Decision Processes. 25

DM Data Manager. 59, 60, 61, 72

DRTDBS Distributed Real-Time Database System. 1, 7, 8, 10, 18, 30, 33, 34, 36, 38, 39, 45, 57, 58, 61, 69, 70, 71, 74, 83, 103, 105, 132

FDR First Deadlock Resolution. 64, 95, 96, 97, 100, 101

FIFO First-In-First-Out. 69, 131

GAg Global Agent. vi, 41, 43, 44, 46, 76

HIAMAP Helper-Initiated Action MAP. 29

LAN Local Area Network. 64

LM Lock Manager. 15, 59, 61

MA Mobile Agent. 35

MAEDD Mobile Agent Enabled Deadlock Detection. 8, 35, 38, 57, 85, 87, 88, 89, 90, 91, 94, 104

MAP Mutual Assistance Protocol. 27, 28

MAP Multi-Agent Platform. 40, 41

MAS Multi Agent System. 5, 6, 7, 8, 10, 23, 24, 25, 27, 29, 38, 39, 43, 76, 103, 104, 131

PCOT Percentage Completed On Time. vi, vii, 82, 85, 87, 89, 90, 91, 94, 96, 97, 99, 100

PDR Priority Deadlock Resolution. 64, 86, 95, 96, 97, 100, 101

POMDP Partially Observable Markov Decision Processes. 25

RIAMAP Requester-Initiated Action MAP. 29

RTDBS Real-Time Database System. 1, 2, 7, 13, 14, 16, 20, 41, 64, 71, 103

S2PL Static Two-Phase Locking. 16

SA Static Agent. 35

SAg Site Agent. vi, 40, 41, 43, 44, 76

TAg Transaction Agent. 47, 49, 50

TG Transaction Generator. 59, 60, 75

TM Transaction Manager. vi, 41, 59, 60, 61, 71, 75

WAN Wide Area Network. 33, 34, 64

WFG Wait-For-Graph. vi, 18, 20, 30, 31, 33, 36, 40, 41, 43, 47, 57, 77, 80, 82

Acknowledgements

I would like to sincerely express my gratitude to my supervisor, Dr. David Casperson, for his invaluable support throughout all stages of this thesis. Such a study would not happen today without his experience, advice, encouragement, and patience.

This Thesis is dedicated to my wife

Nahid Taheri

for her endless love, support, and encouragement.

Chapter 1

Introduction

Database systems are universally used in various applications across almost all industries. In general, these systems provide a safe and efficient method to store and retrieve information. In addition, Real-Time Database Systems are defined as systems where time is a crucial constraint to consider [161]. Database systems can be categorized as centralized or distributed. While a centralized database system stores data at a single node, the Distributed Database System consists of multiple nodes which could be geographically distributed and function independently as data is shared among them through a communication network [48]. Distributed Database Systems provide many advantages over single-node database systems, including higher system availability and throughput, as well as incremental expandability [158]. Distributed Real-Time Database Systems (DRTDBS) are adopted in many real-time applications that require guaranteed response times and stable operation in the case of catastrophic failures, such as banking, robotics, network management and air traffic control systems [4, 164]. The growing trend in keeping

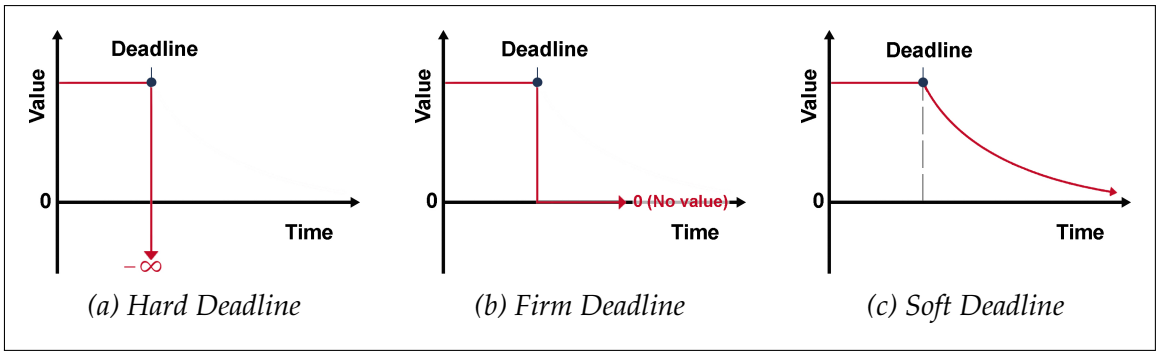
large amounts of data as well as the need for instant access to databases has led to extensive research on DRTDBSs.

Access to a database system is possible through the atomic unit of data processing known as the *transaction*, which carries out the basic operations of requesting or modifying data through the read and write requests; multiple transactions may execute these operations concurrently [124]; however, database transactions must guarantee Atomicity, Consistency, Isolation, and Durability (ACID) [56]. By definition, atomicity protects the completeness of a transaction (that is, prevent from partial execution). Consistency ensures that at the end of any transaction the system is in a valid state, whether it is completed or not. The isolation property guarantees that all transactions are running exclusively independent from one another. Finally, the result of a transaction remains permanent as the consequence of durability [13].

In a Real-Time Database System (RTDBS), "completion of a process or a set of processes has a value to the system which can be expressed as a function of time" [69]. Each transaction has an associated time constraint in the form of a completion deadline. The performance of RTDBS is then evaluated based on the number of transactions that are able to complete their tasks before their deadline expires. The expired transactions are aborted and discarded from the system before the completion of an execution [57].

Typically, transactions have a positive value at arrival time; this value decreases after the deadline expires at a rate depending on the deadline type and the time elapsed. In general, deadlines are categorized as hard, firm, and soft [103]. The value of the transactions with a hard deadline is considered as negative in

cases where they missed their deadline and thus have catastrophic consequences on the system. (Figure 1.1 (a)). An example of a hard deadline would be sensitive operations in nuclear systems.



(a) Hard Deadline *(b) Firm Deadline* *(c) Soft Deadline*

Figure 1.1: Deadline Types

Similar to a hard deadline, a firm deadline is associated with a strict time constraint, but after the deadline expires the value drops to zero (Figure 1.1 (b)). Obermarck described the value of transactions with a firm deadline after the time limit expires as worthless [109]. An example of the firm deadline would be storm forecast systems. In contrast to hard and firm deadlines, a soft deadline provides extra time after the deadline for the transaction to complete their tasks. This interval is called tardy time and the value of the transaction drops gradually to zero at the end of the interval (Figure 1.1 (c)). An example of soft deadline would be Air conditioning system.

1.1 Deadlock Handling

In a database system, there is a potential state when the transaction(s) must wait for a resource held by other transactions. This waiting can be circular because

a transaction may be waiting for a resource locked by another transaction which in turn is waiting for the first transaction. This cyclic wait condition is known as deadlock [32]. No progress can be made until the cycle is broken. Various deadlock handling approaches are discussed in the this section.

There are different approaches to deadlock handling in a database system. These approaches include (a) prevention, (b) avoidance, (c) detection and resolution of a deadlock. Deadlock prevention protocols guarantee a deadlock-free condition [158] by structuring a database system to prevent at least one of the conditions necessary for a deadlock to occur [30] (*i.e.*, "mutual exclusion, hold and wait, no preemption and circular wait" [139]). These algorithms postpone a transaction execution in case of the existence of one of these conditions; the prevented transaction must restart at a later time. Even though implementing a deadlock prevention algorithm is relatively straightforward, the cost of transaction restart is significantly high.

Deadlock avoidance, on the other hand, reduces the system overhead and hence is preferred over the prevention protocols [158]. Deadlock avoidance protocols attempt to predict deadlocks at the time a resource is requested by a transaction and react to the request accordingly to avoid deadlock [30]. Each request is analyzed dynamically in order to achieve efficient transaction execution. Deadlock avoidance algorithms require information about the potential use of each resource associated with each transaction. Unavailability of sufficient information at analysis time will lead the system to inefficient transaction execution.

Contrary to deadlock prevention and avoidance algorithms, deadlock detection algorithms do not preclude the possibility of a deadlock, but attempts to min-

imize its adverse execution impact [24]. The presence of deadlocks is detected by a periodic iteration of a deadlock detection algorithm. When detected, a resolution algorithm selects a transaction to abort, releasing its held resources in order to break the deadlock cycle [24, 30, 154].

It is widely accepted that both deadlock prevention and deadlock avoidance algorithms are conservative and less capable of handling real deadlock problems, because they make unrealistic assumptions about the knowledge of the resource allocation requirements of participating transactions. However, deadlock detection and resolution algorithms are broadly used as an optimistic and feasible solution to the deadlock problem [24, 31, 75, 154]. Implementation of deadlock detection and resolution is possible through a centralized decision-making mechanism or through use of a team of distributed cooperative processes.

1.2 Deadlock Scheduling

The overall performance of deadlock handling protocol in a real-time environment not only depends on the transaction execution cost, but also on how frequently the deadlock handling protocol is executed [75]. In particular, deadlock detection and resolution scheduling is an important factor that can significantly affect the efficiency of deadlock handling [24].

Furthermore, the absence of distributed deadlock detection scheduling, particularly how frequently it should be conducted in a distributed environment, has an insufficient impact on the performance of deadlock handling [24, 149, 154].

In this research, we explored the use of Multi Agent Systems (MAS) to detect and resolve deadlocks.

1.3 Multi Agent Systems

Michael Wooldridge defines an agent as a computer system that is located in some environment (for example in software application or the physical world) and is able to act autonomously in order to fulfill its designated objectives [155]. Although there is no universal definition of an agent, this is the definition we adopted in this thesis. A MAS consists of an environment with several agents interacting with each other while proactively pursuing their objectives. Shoham and Leyton-Brown describe a MAS as a system with a combination of multiple entities, with different objectives or having different data or a combination of both [125].

1.3.1 Teamwork in Agents with Helpful Behaviour

Many tasks in our daily life can only be achieved through human teamwork (that is, the process of working collaboratively with a group of individuals to achieve a specific goal). According to the academic investigation of management practices, mutual support in human teamwork is one of the vital success factors [84]. Predominantly, each of the team partners performs a different piece of the joint task and may mutually perform a cooperative act towards accomplishing the shared goal. For about three decades, consideration of teamwork has become a major field of study in systems using artificial intelligence. The collaborative behaviour

of an agent with a team refers to helpful cooperative action in order to benefit its team. In particular, intelligent agents are able to perform supportive attitudes in a teamwork activity with other agents or humans in pursuance of achieving a joint goal. Fundamental studies of teamwork in MASs have been presented in several studies [27, 35, 54, 88, 116, 143, 155].

The teamwork approach in MAS provides certain advantages for problem solving in distributed environments [133]. In general, multi agent coordination and cooperation as a team offers improvement in a system's robustness and increases flexibility and adaptability [42]. Cooperation and coordination techniques in MAS have different approaches, including centralized, where a single agent assigns tasks to other agents such as in the study by [142], and shared, such as in the study by [131] where tasks are distributed among the agents using negotiation strategies.

In human encounters, having a mutual interaction is intuitively understood, however, this concept needs clarification in agent communication. Once specifications of agents interaction are precisely defined, then it can be improved, formulated, and incorporated into Multi Agent System software development libraries and platforms. Several studies have been carried out on different agents interaction protocols, such as auctions, negotiation, and bargaining [41, 66, 131].

1.4 Problem Statement

RTDBSs add a temporal constraint to the notion of ACID. The increased demand for DRTDBSs requires improved system throughput. Deadlocks are one of the key phenomena which affect performance negatively, particularly when they go undetected, and thus unresolved. Deadlock detection techniques such as the one proposed by Chandy and Misra [21], impose undesirable system overhead. Also, the agent-based deadlock handling algorithms such as [165] are not practical because of including unrealistic assumptions about the network or the significant overhead associated with them.

The problem addressed in this thesis is the potential system performance improvements in DRTDBS by developing a detailed adaptive MAS deadlock detection and resolution algorithm. For this purpose, we study the effectiveness of exploiting a team of agents in the context of DRTDBS, in particular agents with helpful behaviour. Agents have advantage of being on the same site as the peer site, and interacting with the peer locally and autonomously. This allows us to develop algorithms that observe the most up-to-date system information for deadlock detection and resolution and reduce unnecessary communications. We compared our algorithm with existing deadlock detection and resolution algorithms through simulation experiments. Particularly, we investigated the impact of our algorithm on system throughput in a DRTDBS and compared the results with other algorithms including Mobile Agent Enabled Deadlock Detection (`MAEDD`) [17] and Chandy [22]. Note that we chose other algorithms so as to compare our algorithm with another agent-based algorithm, and with a well-known algorithm.

1.5 Thesis Organization

In this thesis, we investigate incorporating a team of agents with helpful behaviour as an aid for deadlock detection and resolution in a distributed environment. We first start with an experiment of using intelligent agents to investigate the transactions' condition at the time of deadlock using an Agent Deadlock Detection Algorithm (`ADDetect`) and then we select the victim transaction judiciously. An Agent Deadlock Resolution Algorithm (`ADRes`), allows agents in the deadlock to deliberately initiate help negotiation by offering to abort its transaction to teammates in order to maximize team benefit voluntarily, and therefore decrease transactions' re-execution costs. Finally, we improve the balance between interaction and bilateral decision-making to leverage the advantage of incorporating agents into a real-time database deadlock situation.

The thesis is organized as follows: Chapter 2 reviews relevant literature in this field of study; Chapter 3 elucidates the Agent Deadlock Combined Detection and Resolution Algorithm (`ADCombine`); Chapter 4 covers the simulator architecture; Chapter 5 lays out the evaluation; and Chapter 6 discusses the conclusions and future work.

Chapter 2

Background and Related Work

This chapter reviews some background studies and related research in Distributed Real-Time Database Systems, deadlock handling, transaction processing, and various execution protocols. Also, we outline some fundamental concepts of Multi Agent Systems (MAS), agent teamwork, helpful behaviour in a team of agents, agent interaction protocols, and bidirectional deliberation on direct help in agent teamwork, with the focus relevant to this thesis.

2.1 Distributed Database Systems

Özsu and Valduriez [112] defined a distributed database as a collection of multiple databases with logical similarities that are spread over a network. A Distributed Database System (DDBS) is a software system that contains a plurality of sites each storing data in at least one database locally and permitting the management of a

distributed database [112, 141]. The following sections provide essential definitions and reference to related works on DDBSs.

2.1.1 Data Distribution

In a Distributed Database System (DDBS), data distribution is possible between sites using replication or partition approaches. In the partitioned approach, there is no intersection between databases at different nodes, and thus unique data items are distributed across distinct nodes. Although the cost of maintaining consistency is reasonably low in partitioned distribution, it is insecure given that system collapse can occur due to failure at a single site. On the other hand, the existence of multiple copies of the same data at different nodes in replicated distribution would guarantee continued progress if one or more sites fail. However, updating all copies of data is necessary to ensure consistency [37]. The locality and replication of data can severely affect performance and integrity of the entire system [12]. Replicated structure can be further classified as fully or partially replicated according to the number of copies of data items in the system. In partial replication, a data item is stored on one or more nodes, whereas, in a fully replicated environment, a copy of each data item is stored at all sites. The variance in the number of replicas for each data item is based on its criticality and access frequency [153]. This is illustrated in Figure 2.1.

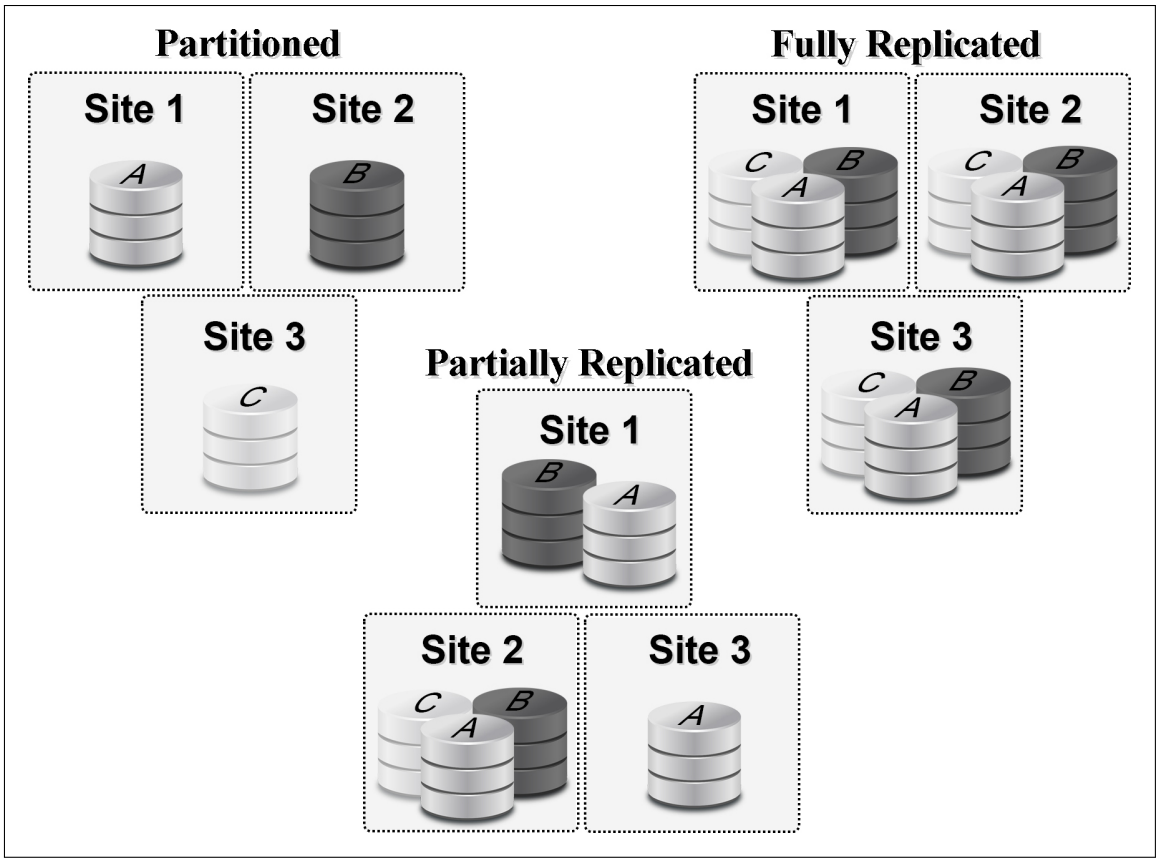


Figure 2.1: Partitioned, fully replicated, and partially replicated data distribution

2.1.2 Transactions

Transactions are the atomic units of data processing in a DDBS [72]. Transactions can be categorized as local or global based on the location of sites necessary to complete the transaction [47]. Local transactions perform tasks only at the site where they originate, whereas global transactions execute at various sites and may involve sub-transactions. In a global transaction execution model, transactions are categorized according to the sites upon which they are executed. The transaction executing at the site of origin is referred to as the master. Sub-transactions, which execute at distributed sites on behalf of the master from a cohort; cohorts must provide stable communication with the master in order to successfully complete a

global transaction [3]. Therefore, successful completion of a transaction in a distributed environment requires extra effort due to the existence of sub-transactions.

The lifetime of each transaction can be divided into two phases: the work phase and the commit phase [40]. In its work phase, a transaction reads or manipulates data. The master process dispatches a cohort of sub-processes, one for each site involved. After the sub-processes have completed their work phase, then respond to the master referring to confirmation of the completed task. When all the cohorts acknowledge their "`WORKDONE`" approval to the master, the work phase of transaction is considered completed. The transaction is then ready to start its second phase. In the commit phase, either a commit protocol records the changes permanently, or an abort protocol executes to discard any adjustment made in the work phase [139].

Execution of multiple transactions on the same data concurrently requires a method that guarantees the serial execution of transactions [12]. Serialization allows the transactions of a RTDBS to be executed simultaneously while maintaining the global order of execution [40]. Serializations of transactions require locking and commit protocols in the work and commit phases, respectively. These protocols are further discussed in Sections 2.1.5 and 2.1.6.

In a RTDBS, each transaction must meet the time constraints assigned to it, as well as the more general constraints of Atomicity, Consistency, Isolation, and Durability (ACID) [40, 138]. Song *et al.* evaluate a RTDBS according to the number of transactions completed before the deadlines [136]. Also, the consequence of missing deadlines and the average lateness or tardiness of late transactions are presented in [136].

2.1.3 Priority Scheduling

Transactions are allocated a priority to specify their degree of importance and order of execution. In the case of data conflict, the priority assignment protocol determines which transactions will be executed first and which transactions will be blocked or restarted. Priority inversion can occur when a higher priority transaction is being blocked by a lower priority transaction [122]. In a real-time database system, priority inversion can cause the undesirable situation of a high priority transaction missing its deadline while waiting for the resource(s) being held by a lower priority transaction.

Liu and Layland classified priority assignment protocols into static, dynamic, and hybrid [92]. In static priority scheduling protocols, a transaction's priority is assigned before the transaction starts its execution in the system. Once assigned, theses priorities do not change [34]. On the other hand, dynamic scheduling protocols assign priorities at run time after considering factors such as deadline, slack time, and execution time. Finally, a priority assignment protocol is said to be hybrid if static priority is used for some transactions and dynamic priority for the others.

2.1.4 Concurrency Control Protocols

Concurrency control is an important method to maintain the consistency of a database management system. A Concurrency Control Protocol (CCP) guarantees that transactions can simultaneously access shared data without interfering with one

another [10]. CCPs ensure the serialization in RTDBS and maintain the atomicity of transactions by controlling the behaviour of the locking and the commit protocols [124]. Thus, the main purpose of employing a CCP is to maximize the concurrency and maintain the consistency of the databases [121]. Some of the best-known proposed CCPs are greedy locking, greedy locking all copies, and adaptive speculative locking [53, 117, 139].

2.1.5 Locking Protocols

Multiple simultaneous access to the same data object can lead to database inconsistency. Therefore, before a transaction or a sub-transaction can access a shared data item, it must to request a lock on that data item. Locks in a database system can be classified as shared and exclusive [74]. Reading a data item can be jointly done with other transactions using a shared lock while modifying data requires exclusive locking. A data item can be held by a single exclusive lock, or by multiple shared locks [126]. When the Lock Manager (LM) receives an access request for a data item from a transaction, it considers the state of the associated lock on the requested data item. If the data is not currently locked exclusively or has a shared lock, then the scheduler gives the lock permission to the transaction. On the other hand, if the data is locked exclusively, then the transaction must wait until the current lock is released and the data item becomes available. This ensures that a data item can be modified by a single transaction at a time. Also, serialization of transactions within the system is guaranteed by applying locks on the data items [73].

In the last decades, there has been research interest in using distributed real-

time locking protocols in order to enhance system performance by improving concurrent execution of transactions [11, 79, 123, 140]. One of the best-known locking protocols, proposed by Abbott and Garcia-Molina in [1], is Two-Phase Locking (2PL) in which execution of a transaction includes two phases of growing and shrinking. The growing phase includes acquiring needed locks to a transaction while during the shrinking phase, the transaction releases its locks; hence, transaction operations execute once all the locks are acquired. The 2PL protocol guarantees data consistency in a database system [12]. The process of a 2PL protocol is illustrated in Figure 2.2.

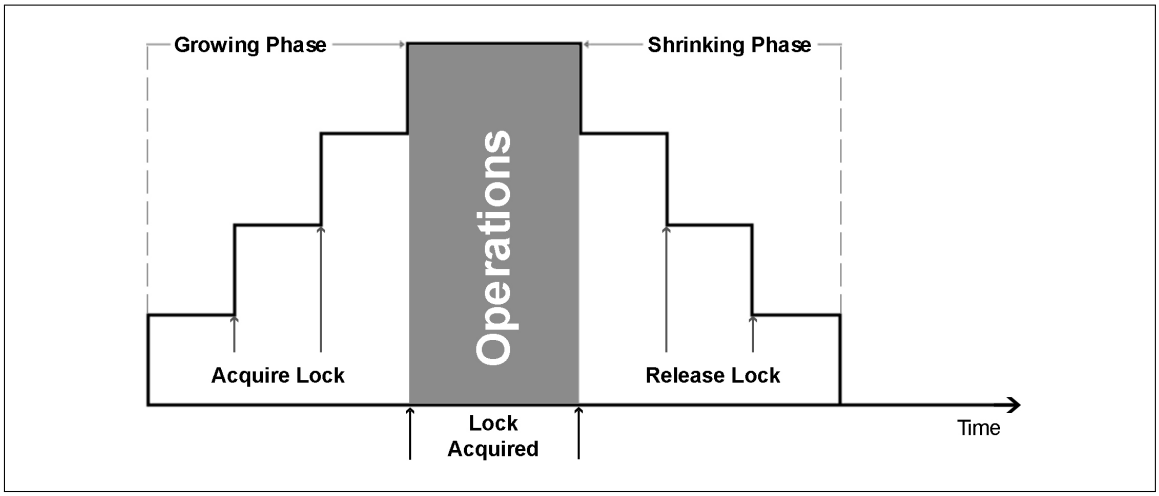


Figure 2.2: Two-Phase Locking Protocol

The 2PL protocol can be divided into Static Two-Phase Locking (S2PL) and Dynamic Two-Phase Locking (D2PL) [80]. D2PL acquires the needed locks for a transaction on demand and releases its aquired locks when the transaction is terminated or committed [146]. In S2PL, the required locks of a transaction are acquired prior to the transaction execution. The locks needed for the transaction during its execution are assumed to be known beforehand in the S2PL protocol [147]. In the last two decades, there has been much research comparing D2PL with S2PL in both

Real-Time and non-Real-Time Database Systems [85, 124].

2.1.6 Commit Protocols

A commit protocol guarantees that the work done by local and global transactions is successfully completed and any modification of data in the database will be permanent. In addition, commit protocols secure the atomic feature of transactions. In order to ensure atomicity of a transaction, commit protocols prevent locks on data items from being released until after the modifications are permanent [130]. Moreover, in order to protect atomicity in a distributed environment, all the operations performed by a global transaction, such as master and cohorts, need to commit individually before the parent transaction can commit. This could possibly result in extensive message passing and logging between master and cohorts, thereby increasing the overall execution time.

In a DDBS, numerous commit protocols have been proposed [40, 58, 59, 102, 130]. One of the commit protocols that is most commonly used in practice is the Two-Phase Commit (`2PC`), proposed by Gupta *et al.* [55]. The `2PC` protocol have two main states of the *prepare phase* and the *commit phase*. At first, the preparation phase starts when all cohorts send a "`WORKDONE`" message to the master; in other words, when the cohorts successfully execute their assigned tasks and respond to the master in a transaction completion report. Soon after collecting that information, the master initiates committing by sending a "`PREPARE-TO-COMMIT`" message back to all of the cohorts. Afterward, each cohort prepares a log regarding committing or aborting the execution and replies to the master by voting to commit or to abort. Meanwhile, prepared-to-commit cohorts await further notice from the

master. The commit phase starts when a master has collected all of the feedback from cohorts and unilaterally makes a global decision to either commit or abort the global transaction. In the case of a unanimous commit decision, the master creates a commit log and broadcasts the commit message to all cohorts. Each cohort then reacts based on the received command by logging commit and abort operations for the global commit and global abort respectively. Finally, both the cohort and master reach the final state of ABORT or COMMIT, and the transaction is considered complete, and the lock on the modified data can be released [55]. A diagram of a Two-Phase Commit protocol is presented in Figure 2.3.

2.1.7 Deadlocks

The deadlock problem occurs in many different contexts [65]. A universal example is the traffic deadlock when four cars arrive at a four-way intersection at the same time, each wishing to proceed ahead. Deadlock is a common problem within multitasking concurrent programming systems [30]. Furthermore, the deadlock problem becomes more sophisticated where the underlying system is geographically distributed and the transactions have time constraints, in particular, DRTDBS [61]. The problem of deadlock handling has received much attention in DDBS literature [17, 21, 30–32, 75, 77, 109, 129, 133, 154, 158, 159, 165].

A deadlock in DRTDBS occurs when a set of transactions are not able to complete their tasks because each transaction is waiting for the resource held by other transactions from this set [77]. A deadlocks drive DRTDBS into an undesirable situation that affects the system negatively [30]. The consequences of deadlocks on the system include decreasing the system throughput, collapsing the utilization of

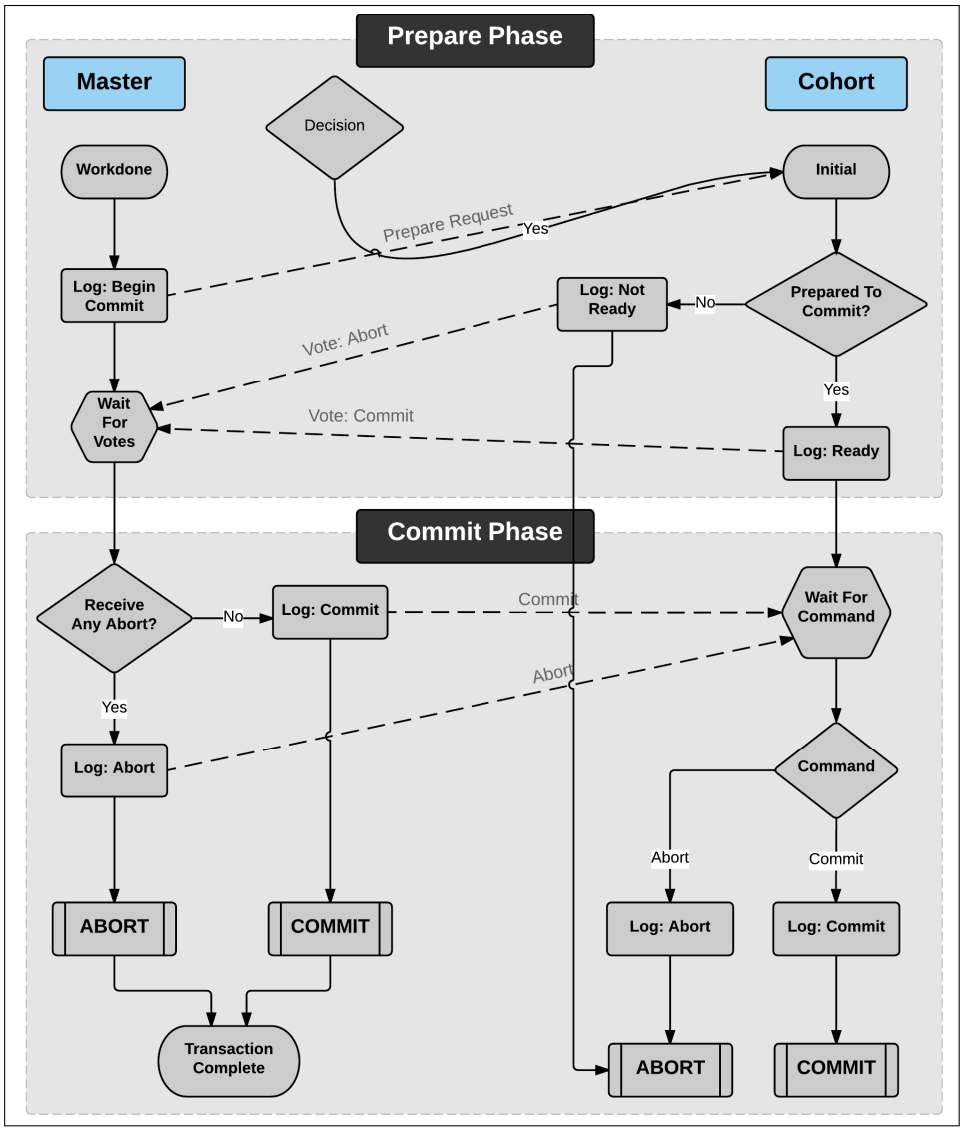


Figure 2.3: Two-Phase Commit Protocol

the involved resources to zero, and increasing the number of deadlocked transactions with deadlock persistence time [128]. The deadlock cycles[1] do not terminate without outside interference [30]. The transaction waiting dependencies is called a Wait-For-Graph (WFG). A WFG consists of a set of Transactions $\{T_1, T_2, \cdots, T_n\}$ where each (T_i, T_j) denotes that T_i is waiting for a resource held by T_j [157]. An example of a circular deadlock state and its resource allocation graph is illustrated in Figure 2.4 (a) and its WFG represented in Figure 2.4 (b).

[1] In a finite system, deadlocks contain a finite loop.

One of the conditions necessary for a deadlock to occur is no preemption [139]. Deadlock detection and resolution algorithms can be used in the absence of a preemption protocol. The details of preemption are presented in the next section.

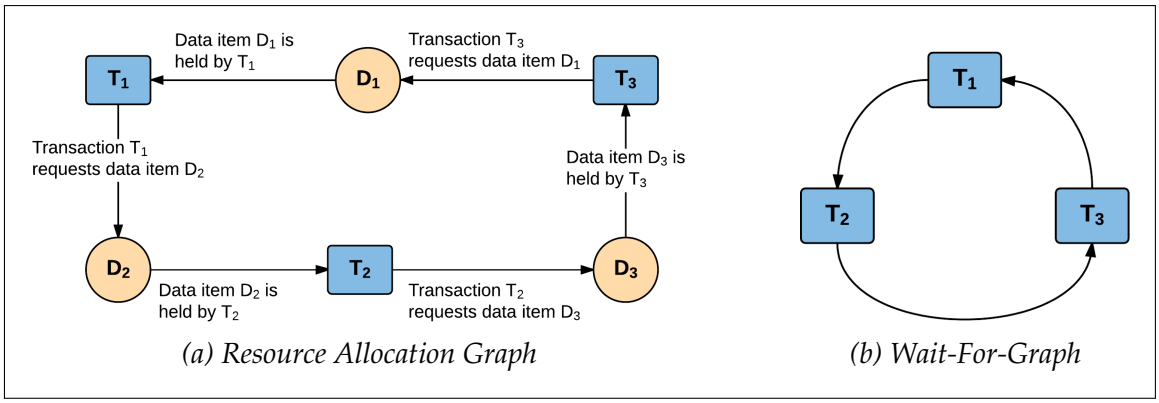


Figure 2.4: a) Transaction T_1 is holding data item D_1 and requests data item D_2 which is held by T_2 while T_2 is waiting for T_3 to release data item D_3 and T_3 will not release D_3 until it acquires D_1 and thus the system is in a deadlock; b) The structure of a WFG

2.1.7.1 Preemption

Preemption of one or more low priority blocking transactions may become necessary to avoid blockage of higher priority transactions [81]. Priority inversion occurs when a high priority transaction is blocked by a lower priority transaction. This can be alleviated by priority inheritance and priority ceiling protocols. Priority inheritance declares that when a higher priority transaction T_H is blocked by a lower priority transaction T_L, T_L inherits the priority of T_H temporarily. Once the execution is completed, T_L restore its initial priority and releases all the locks that were causing the priority inversion [115]. Even though the priority inheritance protocol may reduce blocking time of T_H in general, it cannot assist in a deadlock situation [122]. Moreover, T_H can be stuck in a chain of lower priority transactions that would significantly increase the waiting duration [82].

A priority ceiling protocol is an extension of the priority inheritance protocol. In this protocol, the highest priority of the transactions trying to access the same data item is assigned to the transaction causing the priority inversion. Therefore, preempting the blocking transaction requires a priority, superior to other preempted transactions. Despite the fact that a priority ceiling can reduce blocking time and prevent a deadlock from occurrence [122], it may cause unnecessary blockage of a lower priority transaction by a higher priority transaction while reading [160]. Additionally, in a RTDBS which is using the priority ceiling protocol, serialization of real-time transactions cannot be guaranteed [82].

Huang *et al.* proposed the concept of conditional priority inheritance based on transaction execution time estimation [62]. In the case of priority inversion, newly arrived high priority transaction evaluates waiting time required for the lower priority transaction to be completed. If the higher priority transaction can afford to wait without missing its deadline, then the lower priority transaction inherits the priority of higher priority transaction. Otherwise, the lower priority transaction is preempted. Conditional priority inheritance manages transaction abortion in such a way that the system not only benefits from reducing wasted resources, but also avoids extended blocking time for high priority transactions [151]. Despite the advantage of conditional preemption in a one-to-one conflict situation (that is, there are no multiple transactions involved in the conflict), the protocol may fail in a chained block. Also, the higher priority transaction should have an accurate time estimate to perform properly, which may not always be available [151].

2.1.7.2 Deadlocks Persistence Time

The persistence time of a deadlock denotes the temporal interval between the time at which deadlock detection and resolution algorithm detects and resolves the deadlock and the time at which the deadlock is initially formed. It grows linearly until the deadlock is resolved using one of the deadlock resolution algorithms [24]. Park *et al.* discussed that upon initiation of a deadlock, other transactions requesting resources currently held by the deadlocked transactions immediately fall into the deadlock state [113]. Therefore, a formed deadlock acts as a transaction trap that may increases the deadlock size. Particularly, increasing the deadlock persistence time results in a growing number of blocked transactions which leads to a higher deadlock resolution cost [129]. Note that the deadlock detection process time is significantly more than a resolution process time. Hence, the deadlock persistence time can be stopped as soon as the deadlock detection algorithm executes.

2.1.7.3 Deadlocks Detection Interval

The deadlock detection interval represents the time between each search for deadlock. When the detection interval is small, the deadlock detection algorithm executes more frequently result in reducing deadlock persistence time. On the other hand, Ling *et al.* showed that an optimal deadlock detection interval can minimize the deadlock persistence time and increase the system throughput [24, 91].

2.2 Intelligent Agents

To consider an agent (see also 1.3) to be intelligent, the agent is expected to hold three capabilities, the ability to be: *reactive*, *proactive*, and *social* [156]. Reactive behaviour is defined as the ability to perceive the environment and respond accordingly to changes in a timely manner. Proactive behaviour is characterized by the ability to use creativity in executing goal-directed actions. Finally, social behaviour is explained as the potential of having interactions with other agents and possibly people.

Russell and Norvig have proposed a classification model for different types of environments based on where an agent might be located [118]. For example, an environment might be *static* or *dynamic*. A static environment can be considered unaffected unless by the agent's action. However, a dynamic environment is beyond the agent's control because other processes are operating on that as well. Likewise, an environment might be *deterministic* in which any action performed by an agent has a single guaranteed effect, and there is no uncertainty about the state of environment derived from that action; or it can be *non-deterministic*. In a Multi Agent System (MAS), the environment that an agent is situated in has some initial state as perceived by an agent. The agent then performs an action in order to achieve its objective; meanwhile, the environment may transform to another state because of an agent's action. Note that changing the state of an environment could cause the agent to fail in achieving its goal, potentially requiring it to execute another response [155].

The *practical reasoning* architecture proposed from philosophical work by Bratman [14] is known as one of the most reliable agent designs of the last two decades.

In general, practical reasoning agents are equipped with a mental state similar to humans' minds, which is then used in their decision-making process. In contrast to the practical reasoning that is directed toward actions, *theoretical reasoning* is directed towards beliefs. Theoretical reasoning agents consider logical theorems as models of decision-making while practical reasoning agents employ *deliberation* and *means-ends reasoning* for their decision-making process. Deliberation indicates *what* state of affairs to achieve, whereas means-ends reasoning identifies *how* to achieve these states of affairs. The output of means-ends reasoning is the *plan* and the output of deliberation is the *intention* in which the agent has some level of commitment [155].

The most popular framework for designing practical reasoning agents is known as *Belief-Desire-Intention (BDI)*. The performance of a MAS with a BDI framework relies on the specified mental attitudes [15]. In this context, agents build their informative state (beliefs) perceived about the environment which may not necessarily be accurate. Also, agents apply these beliefs to establish their motivational state (desires) that indicates the states of the environment that the agent would like to accomplish. Finally, selecting particular tasks from desires leads to the deliberative state of a rational agent (intention) to which the agent is committed [155]. Overall, the BDI structure has been implemented and used in many real world systems [49, 50, 93, 99].

2.2.1 Teamwork

Teamwork involves each artificial intelligent agent's commitment to a joint action in order to accomplish a shared goal [27]. Over the past two decades, there have

been many scientific studies on the semantics of agent teamwork. The fundamental investigation presented by Cohen *et al.* [26] defines the concept of joint intention, joint action, and joint commitment [88]. They investigate the necessity of multiple joint actions involving agents when individual agents are sharing certain specific mental properties. Also, they study joint commitment and joint intention specifications when a team of agents acts as an aggregate agent [88].

In a highly dynamic environment, where the state of the environment is frequently changing, finding an optimal decisive goal-directed teamwork policy is complicated since there is tension between individual intentions and acting as one aggregate agent [27]. Kaelbling *et al.* work on Partially Observable Markov Decision Processes (POMDP) which allow an agent to make decisions when faced with uncertainty, especially in a highly dynamic environment [70]. Also, in a probability distribution over all possible states that is given to individual agents, both actions and perceptions with uncertainty are considered in this model. The implementation of POMDP is applied to real world domains, including robot navigation systems [110] and multi agent teamwork research studies [105, 106, 148]. Furthermore, decentralized POMDP (DEC-POMDP) provides frameworks to model POMDP in MAS [9].

Providing the ability to perform optimization in a distributed environment verifies a powerful framework for multi agent teamwork, called a Distributed Constraint Optimization Problem (DCOP), as investigated by Mailler and Lesser [95]. In the proposed framework, agents are cooperative in teamwork actions and share a common reward function. According to the research done by Taylor *et al.*, execution-time reasoning is considered a critical component of MAS and thus DCOP is applied to address deadline problems, task allocations, and coordinating teams of

agents [148].

2.2.2 Helpful Behaviour

Agents are situated in an environment where unpredictable states and unexpected results may occur beyond the agent developer's expectations. Furthermore, implementation of an agent with a lot of capabilities could be an expensive task. Hence, agent developers prefer to construct effective agents in terms of ability in their objective domain, so as to reuse them in similar tasks. Although an agent may be designed for a certain application, it could possibly need assistance from its teammates that can provide a positive impact on team performance [114]. Generally, agents with helpful behaviour are equipped with the capability of assisting other team members by performing some action or providing relevant information. In the last two decades, helpful behaviour in agent teamwork has been received much attention [18, 67, 71, 96, 100, 114]

Even though enough encouragements exist for an agent to help its teammates achieve a committed shared goal, still some level of deliberations need to be considered. Therefore, Kamar *et al.* discuss the existence of some cost in agent's interaction, which may influence the team's expenses [71]. There are various source of costs, such as communication coste, spent resources whilst helping, and missing opportunities to execute other actions. According to work presented by Kamar *et al.*, help in agent teamwork includes either executing other's actions or providing some relevant information for a teammate [71]. Moreover, they use a unilateral decision-making system which means that help deliberation is made by only one member of the team. However, because agents use their beliefs as the only source

of help in decision making, accuracy is not guaranteed.

In a MAS, helpful behaviour can be enforced by a centralized mechanism over a group of agents, or it can be specified as direct help. During agents' teamwork, direct assistance can be initiated by an agent as needed and bilaterally decided by the agents involved in the help act [54]. Research in specifying direct help as one of the components in a decentralized MAS has been investigated in [35]. Furthermore, Nalbandyan introduced the Mutual Assistance Protocol (MAP), which implements a direct help mechanism in a team of agents with different skills, situated in a dynamically changing environment [107]. The proposed algorithm is based on a bilaterally distributed agreement in which both requester and helper agents agree to execute a helpful action. Notably, the research argued that bilateral decision-making has a behavioural advantage over the unilateral approach when the communication cost is low or mutual knowledge between agents is high.

Malek Akhlagh extended this work and proposed an interaction protocol for bilateral direct help, called the Bidirectionally Initiated Action MAP (BIAMAP) [96]. The research is based on a mutually distributed agreement in a distributed environment where both the requester and helper engage in a shared decision. Additionally, this research illustrates some improvement over the unilateral approach when the communication cost is comparably low.

2.2.3 Agent protocols

In general, agent protocols can be categorized into two groups, hi-level protocols (*interaction protocol*) and low-level protocols (*communication protocol*). Interaction

protocols model individual agents' social behaviour, that is, coordination, negotiation, and collaboration, by implementing an anthology of rules. However, the Contract Net Protocol (CNP) introduced by Reid Smith [131] is one of the most successful and widely used agent interaction protocols in a distributed environment. On the other hand, agent interaction protocols rely on communication protocols that control the manner of sending and receiving information between agents. Accordingly, communication protocols provide speech act classification and the logic for the agents to understand each other while interacting [155]. Overall, two famous agent communication protocols, FIPA-ACL and KQML/KIF, have been proposed by the Foundation of Intelligent Physical Agents [43] and Mayfield *et al.* [98] respectively.

The agent's social behaviour might vary in terms of properties. Hence, the theory of designing a universal interaction protocol among agent teams is not realistic, according to Dunn-Davies *et al.* [36]. Basically, they argue that each domain requires constructing specific interaction protocols for individual agents.

2.2.4 Action Help

Bidirectionally Initiated Action MAP (BIAMAP) was implemented for reciprocal help behaviour in agent teamwork [96]. The proposed method is an extension of the Mutual Assistance Protocol (MAP) [107] in which agents are equipped with the capability of assisting their teammates directly when the mutually distributed agreement is possible through a bidding sequence. A bidding sequence is defined as a situation when an agent receives the task announcement, then evaluates it according to some factors. If the agent realizes that it is suitable for the task, it

submits a bid [107]. Overall, BIAMAP provides the significant feature of providing and receiving help while sending or requesting for help.

In another approach, different interaction models toward help actions are used, such as Requester-Initiated Action MAP (RIAMAP) and Helper-Initiated Action MAP (HIAMAP) [97, 107]. In the first model, agents are able to proactively receive broadcasted help requests from teammates, whereas in the second model they are capable of proactively offering help to teammates. It has been demonstrated that performance of RIAMAP and HIAMAP are complementary in a MAS [108].

Applying cooperation and coordination in geographically Distributed Artificial Intelligence (DAI) systems with limited resources was investigated by Findler and Elder [42]. In this research, resource conflict is managed by implementing a technique called "hierarchical iterative conflict resolution" [42]. In the proposed technique, agents are situated in a highly dynamic environment where agents' tasks are prioritized based on CNP negotiation between groups of agents in order to achieve a common goal. When tasks are constrained by time, agents with a higher priority can borrow or take resources from lower priority agents and solve their task earlier. The author argues that employing helpful agents with cooperation and coordination behaviour can be useful in other distributed applications where limited resources have to be allocated to different processes on the basis of the urgency and importance of the tasks at hand.

2.2.5 Agent Memory

Lerman *et al.* proposed agent memory as a general mechanism for adaptation of artificial agents [87]. By employing memory into BDI architecture, agents can modify their behaviour based on past environmental events. Each individual agent uses memory, which is constructed based on its perception of the environment, to estimate the global state of the system and adjust its actions accordingly.

2.3 Deadlock Detection and Resolution Algorithms

Generally, deadlock detection and deadlock resolutions in DDBSs are discussed separately; however, the latter is as important as the former. In this section, we provide a summary of existing distributed deadlock detection and deadlock resolution algorithms.

2.3.1 Types of Deadlock Detection

Distributed deadlock detection algorithms can generally be divided into four major categories: *Path-Pushing (WFG-based)*(see Figure 2.4), *Edge-Chasing (Probe-based)*, *Diffusing Computation*, and *Global State Detection* [75]. The first category includes the most common algorithms adopted in DRTDBS. A deadlock detection algorithm must satisfy two main criteria: First, no false deadlock detection and second, all the deadlocks must be detected in a finite time [6].

2.3.1.1 Path-Pushing

Path-pushing algorithms support an explicit table of waiting a dependencies in the form of a directed interdependency graph, that is, a WFG. Each site creates its local waiting processes list periodically in order to create a local WFG. Thereafter, every site sends its local WFG to other sites in the system. The integrated topology of the waiting process in the distributed system is shared between all sites once all the local WFGs are collected from each site. Then, each site updates its local WFG by inserting the received wait dependency to develop a global WFG. This detection process is finalized by detecting any existing deadlock in the generated global WFG. One of the best-known algorithms in this category was proposed by Obermarck [109].

In the Obermarck [109] algorithm, each site constructs its local WFG and receives the WFG from other sites using a distinguished special virtual node called *External*. The External node $(\mathtt{Ex})$ represents the portion of a global WFG that is unknown to the site. The local WFG is then updated accordingly and the presence of any deadlock is checked. The found deadlocks are resolved by breaking cycles that do not contain an External node. For the deadlock cycles which contain the External node in the form of $\mathtt{Ex} \rightarrow T_1 \rightarrow T_2 \rightarrow \cdots \rightarrow T_n \rightarrow \mathtt{Ex}$ which possibly constitute a global deadlock, the current WFG is sent in the form of a "string" message to the site that T_n resides in if and only if the ID of T_1 is greater than the ID of T_n. In this way, the number of sending messages is reduced [78]. Note that $\mathtt{Ex}$ represents a *possible* wait condition, which need not exist [76]. The algorithm suffers from false deadlock detection as the constructed asynchronous WFG does not represent a snapshot of the global WFG at any instant [6]. An example of the false deadlock detection process using the Obermarck algorithm is given in Fig-

ure 2.5. In this example, T_1 and T_3 originate at site 1 and wait for T_2 to complete its tasks, which resides on site 2. However, T_2 is waiting for the resources blocked by T_1 and T_3. When a deadlock detection has started, site 1 will send the WFG path $Ex \rightarrow T_2 \rightarrow T_1 \rightarrow Ex$ to site 2, whereas site 2 will send $Ex \rightarrow T_3 \rightarrow T_2 \rightarrow Ex$ to site 1. When the WFG is transmitted, site 2 could abort T_2 to break the cycle $T_1 \rightarrow T_2 \rightarrow T_1$ while site 1 could abort T_3 to resolve the deadlock $T_3 \rightarrow T_2 \rightarrow T_3$, even though this deadlock does not exist anymore.

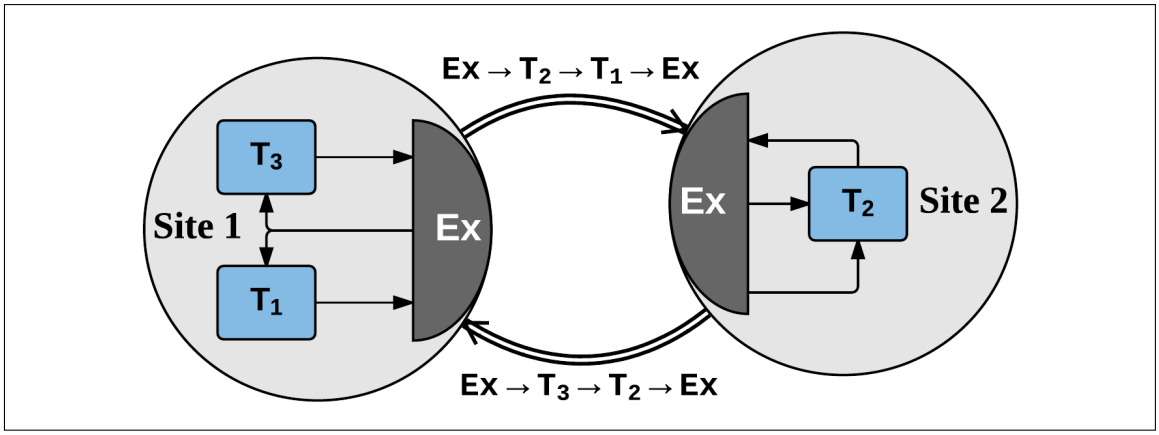


Figure 2.5: False Deadlock Detection in Obermarck's Algorithm

2.3.1.2 Edge-Chasing

Edge-chasing algorithms use a special message called a probe to detect deadlocks without constructing an explicit representation of the graph. The deadlock detection process is initiated by a process transmitting probes to the processes holding the resources it is waiting for. A process that receives the probe message broadcasts it to all the processes it is waiting for. If the initiator process receives a probe sent by itself, it can announce a deadlock because the probe must have traveled a cycle. Note that each probe message contains information to be identified by its initiator.

Probe-based algorithms were originally proposed by Chandy and Misra [21].

In the algorithm proposed by Chandy *et al.* [22], a probe message is initiated by one transaction sent to another for deadlock detection. The probe message is a 3-word message length in the format of (i, j, k) where i is the initiator transaction, j is a transaction that i is locally waiting for, whilst k is the transaction that holds a resource that j is waiting for which resides in a different site than i and j. An example of a deadlock detection process using the Chandy *et al.* algorithm is presented in Figure 2.6. Suppose a deadlock detection is initiated by T_1 in Site 1. On this site, T_1 is waiting for the data held by T_2, which is waiting for the blocked data by T_3 and T_3 is waiting for T_4 to release its blocked data. Meanwhile, T_4 is waiting for the data held by T_5 in Site 2. The global probe message (T_1, T_4, T_5) will be sent to T_5. Then, T_5 broadcasts the probe until it is received by the initiator (*i.e.*, (T_1, T_{14}, T_1) resulting in detecting of a deadlock).

2.3.1.3 Diffusing Computation and Global State Detection

Diffusing computations is proposed by Dijkstra and Scholten in [33] and global state detection is proposed by Chandy and Lamport in [19] is used to detect and terminate deadlocks in a distributed system. These deadlock detection algorithms have found use in distributed simulation [20]. In this thesis, we only use algorithms of type path-pushing and edge-chasing.

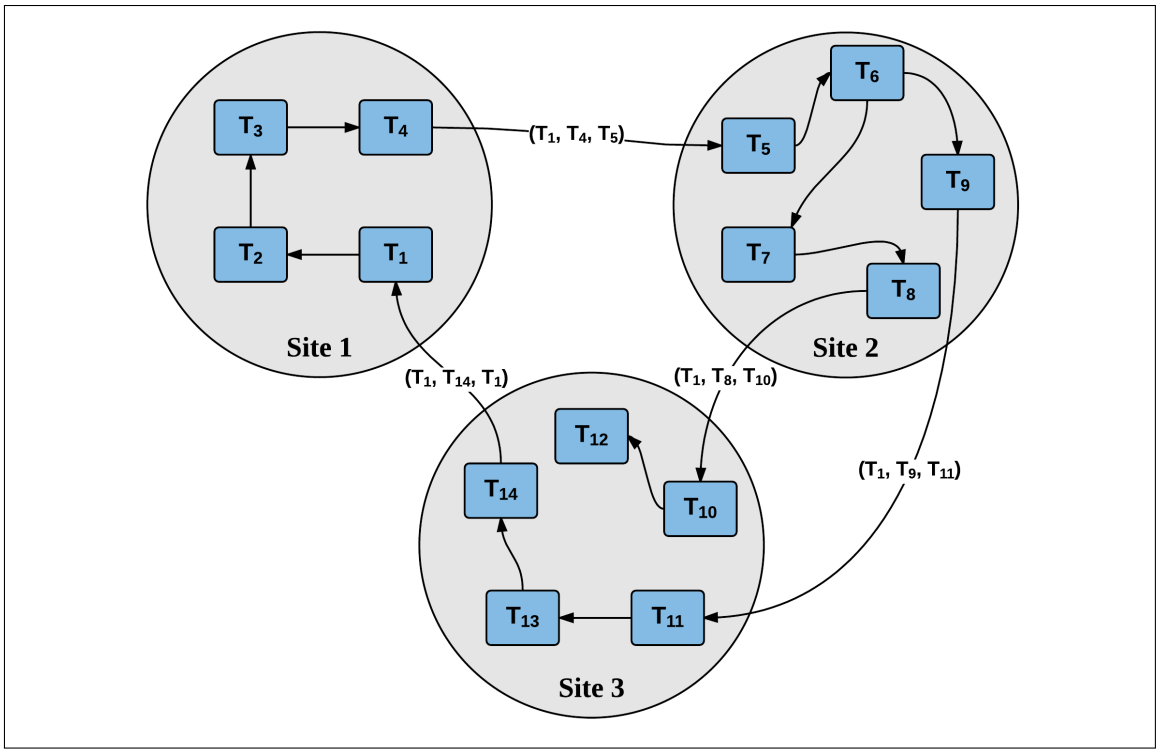


Figure 2.6: Chandy Mirsa Haas Algorithm

2.3.2 Multi Agent Deadlock Detection

In traditional deadlock detection approaches, the deadlock detection algorithms need to be integrated into the server code. Employing intelligent agents as deadlock detectors in a distributed system allows the DRTDBS developers to develop a flexible architecture by having a team of agents to to detect and resolve deadlocks which are functionally separate from the server application logic [17, 135, 165]. Recently, the multi agent technology provides a new approach for structuring and coordinating a Wide Area Network (WAN) and distributed services that require intensive real-time interactions [16, 45]. The main idea of using deadlock detection agents in this context is to distribute the information about the transactions' dependencies, global WFG in particular, between different agents [76].

There are two different approaches to the cooperation in handling deadlocks, whether to detect or resolve cycles: An agent can migrate through the system to collect needed data. Alternatively, an agent can reside on a permanent site to provide support remotely as the need arises. The former approach is known as a *mobile agent*, and the latter is a *stationary agent*. The practical impact of the two different approaches on database system throughput has been investigated through simulation studies [17, 42, 165]. While mobile agents are suitable for distributed environments that require intensive real-time interactions [16], stationary agents often perform better in cooperating and coordinating in a distributed Wide Area Network and highly dynamic environment with limited resources [38]. The advantage of stationary agents has given rise to the research question about the possibility of improving the throughput of DRTDBS by using stationary agents with helpful behaviour.

Cao *et al.* proposed a new algorithm called Mobile Agent Enabled Deadlock Detection (`MAEDD`) [17]. In this algorithm, Static Agents (SA) reside on each site. SAs initialize and dispatch Mobile Agent (MAs). Each MA is in the type of mobile agents, and SAs are in the type stationary agents. The SA is responsible for managing a resource location table which records where the available resources are. In each deadlock detection interval, each SA dispatch a MA which encapsulates the deadlock detection strategy, together with the necessary data, and then dispatches it to the network. MAs are capable of navigating through the network to exchange information and perform tasks at the sites they visit. Once a deadlock cycle is detected by one of the MAs, it will select a victim process according to the embedded deadlock resolution algorithm.

Zhou *et al.* [165] proposed the M-Guard algorithm for deadlock detection in

distributed systems under the MAEDD framework. M-Guard is an agent roaming in the system in order to collect resource requests or propagate information for detecting deadlock cycles as well as propagating the resource information among the sites. Employing M-Guard as an agent with a dual role reduces the overall communication overhead and deadlock cycle persisting time when the scale of the whole system is not too big. However, M-Guard is insecure given that the algorithm collapse can occur due to failure of the agent.

2.3.3 Deadlock Resolution

In a DRTDBS, a deadlock resolution algorithm must be triggered soon enough to permit the deadlocked transactions to complete within their deadline after being restarted [159]. Nonetheless, resolving a deadlock by indiscriminately aborting deadlocked transactions is significantly inefficient because dropping transactions with a higher priority have a negative impact on the system [24]. Aborting all the deadlocked transactions is extremely costly because computations have to start over again. Also, a transitive blocked transaction might not belong to any deadlock cycle in the WFG [24, 75, 149].

Distributed deadlock resolution algorithms must know all the transactions and resources needed by the transactions to resolve a deadlock efficiently [24]. The minimum abort set problem in a deadlock resolution algorithm is to determine a set of victim transactions whose abortion resolves the deadlock to minimizes the overall abortion cost [149]. The cost involves a transaction's partial rollback, lock de-escalation, or the transaction's complete termination [120]. Although the overall cost of deadlock handling is closely associated with aggregated deadlock detec-

tion and resolution cost, deadlock resolution can be more expensive than deadlock detection regarding message complexity, particularly when the deadlock size increases. Park *et al.* [113] pointed out that "the reduction of deadlock resolution cost can be achieved at the expense of deadlock detection cost."

In general, deadlock resolution in a database system involves the following steps: victim transaction selection, victim transaction abortion, and deletion of all the deadlock detection information concerning the victim transaction at all sites [89, 134, 137]. After the victim transaction is selected, a deadlock resolution algorithm must terminate the transaction and release all the resources held by it. Execution of the first two steps, *i.e.*, selection and abortion of the victim transaction, can be computationally expensive for the optimal resolution of the deadlock. Particularly, in an environment where a transaction can concurrently wait for multiple resources and the allocation of a released resource to another transaction can cause a deadlock. The last step, *i.e.*, deletion of all the deadlock detection information, is even more critical, since deleting the information related to the victim transaction can be delayed which may cause several other transactions to wait, leadings to a false deadlock [30]. An example of a traditional deadlock resolution algorithm is Priority Deadlock Resolution [25], which selects the transaction with the lowest priority as the victim.

Chapter 3

Proposed Deadlock Handling Model

In the previous chapter, we saw that deadlock detection and resolution are important modules of practical Distributed Real-Time Database System (DRTDBS). This chapter motivates using agent-based deadlock detection and resolution, and presents new algorithms to do so. Section 3.1 describes the research motivation. Section 3.2 formalizes the deadlock detection and resolution model by explicitly presenting the details of each algorithm. Section 3.2 then presents a combined algorithm.

3.1 The Motivation and Research Objectives

Our motivation for employing of agents in a Distributed Database System (DDBS) emanates from agents' potential benefits in practical applications. In this study, we are interested in investigating the actual impact of agents' teamwork upon system

throughput. Our research objectives involve developing a comprehensive mechanism for the deadlock detection and resolution process in transactions processing in order to increase system throughput. The Mobile Agent Enabled Deadlock Detection (MAEDD) [17] approach provides a Multi Agent System (MAS) mechanism by enabling agents in a database system to detect deadlock cycles, and then resolve the deadlocks (as discussed in Section 2.3.2). Distributed planning on a shared goal is the underlying principle of MAEDD. From an individual perspective, the members of an agent team follow their beliefs to jointly deliberate on performing an action. Each action results from either a multi distributed agreement or a unilateral decision that is in the interests of the team and thus the whole system.

In this research, we propose a new model that uses a team of stationary agents with helpful behaviour to implement distributed deadlock detection and resolution. The specifications of this new algorithm are intended to leverage the advantages of agents' cooperation and coordination in handling deadlock cycles.

Several publications on the MAS approach to distributed deadlock detection and resolution demonstrate that the analysis of the impact of stationary agents on system performance is appropriate for our comparative studies of new and existing algorithms. The impact of the new algorithm on DRTDBS completion rates and throughput is explored through simulation experiments in Chapter 5.

3.2 Deadlock Detection and Resolution Model

In traditional deadlock detection approaches in distributed systems using either path-pushing or edge-chasing, the algorithm must be integrated into the server code. We consider instead a group of stationary agents resides on server sites in a distributed system. This allows us to develop a flexible architecture by having stationary agents observe the status of the system and coordinate with other agents, which are functionally separated from the application logic. Using the MAS is not restricted to a particular database system, and can potentially be applied to different practical system applications [17]. In the following sections, we outline the basics of the distributed deadlock detection and deadlock resolution algorithms using a team of artificial agents and formulate the questions to be explored in the rest of this thesis.

3.2.1 Agent Deadlock Detection Algorithm

In this section, we first describe the implementation details of different agents in our deadlock detection algorithm, `ADDetect`. Then we present the deadlock detection model and agents interaction. Finally, we introduce an improvement to the algorithm.

Site Agents (SAgs) are capable of observing the state of a site and performing tasks. SAgs reside on the Multi-Agent Platform (MAP) of each site. The SAg is responsible for holding transaction wait dependencies (that is, the local Wait-For-Graph (WFG)) for a site. When a local deadlock is detected, the SAg reacts by

dropping a transaction in the cycle to release the resources it holds. Each SAg conducts its belief base through perception. Figure 3.1 illustrates the proposed framework of the Agent Deadlock Detection Algorithm (`ADDetect`).

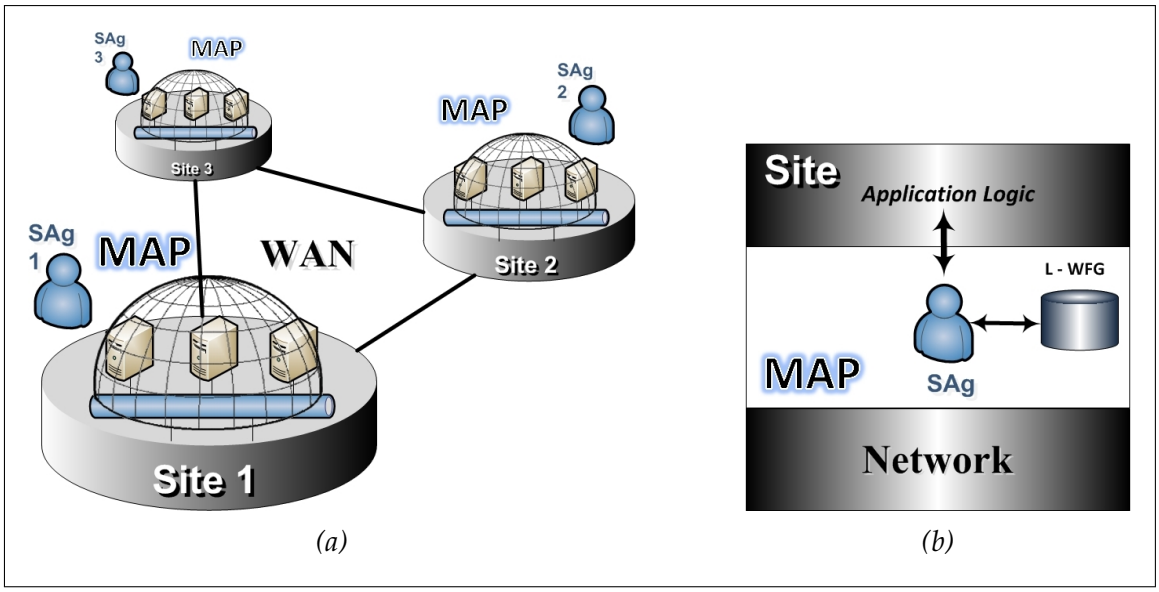


Figure 3.1: The Framework of `ADDetect`

Global Agents (GAgs) are invoked and informed by SAgs and reside on the same site MAP. A GAg interacts with the SAgs of each site to exchange information (discussed in Section 3.2.2). Multiple cooperative GAgs can work together in a deadlock detection process. If necessary, more GAgs can be dispatched depending on the current status of the system. When a GAg detects a global deadlock, it selects a transaction to abort by a deadlock resolution algorithm. The SAg corresponding to the victim transaction is informed to terminate the transaction, thereby breaking the deadlock. Each GAg conducts its belief base through communication and acts rationally in the interest of the global agents' team. Note that a Real-Time Database System (RTDBS) is considered a dynamic environment. Particularly, the state of a server site can be changed by events other than agents' actions. Figure 3.2 illustrates the interaction between a site agent and a global agent.

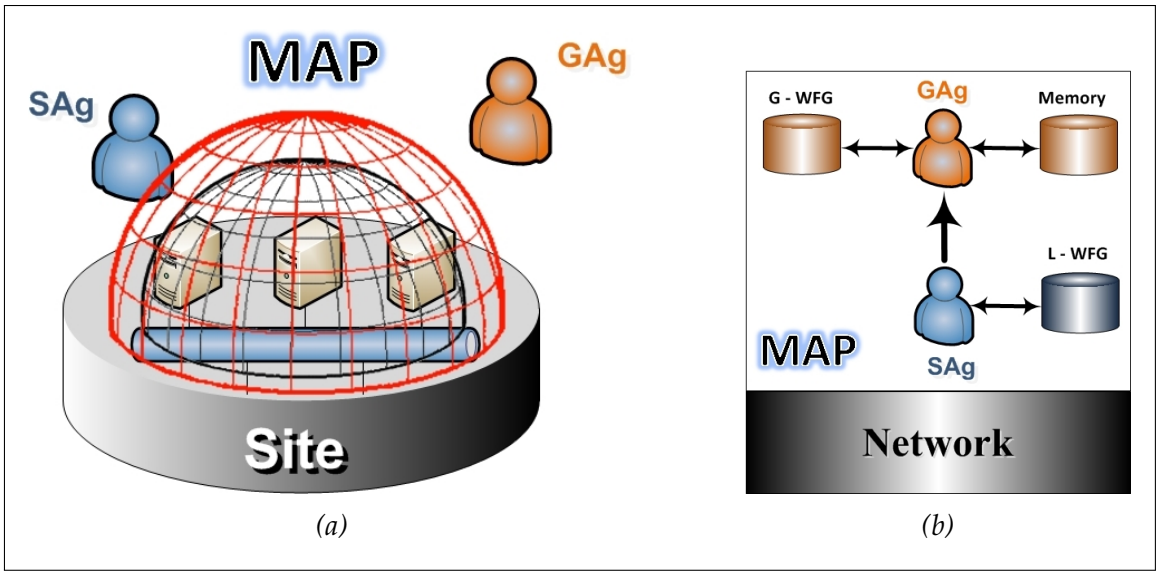


Figure 3.2: The Interaction Between SAg and GAg

3.2.2 Agent Deadlock Detection Model

In the `ADDetect` approach, every site accommodates a site agent responsible for maintaining a local WFG for that site. The maintained WFG is a list of entries, and each entry describes a dependency edge (as explained in Section 2.1.7). The WFGs are in the form of directed graphs in which the transactions are presented as vertices, and an edge from transaction T_i to T_j implies T_j is holding a resource that T_i needs and thus T_i is waiting for T_j to release its lock on that resource. Let $\{SAg_1, SAg_2, \cdots, SAg_n\}$ denote the set of site agents of n sites in our DDBS. From time to time, each SAg_i gathers the state of all local processes of site i to create an up-to-date local WFG. Then the SAg_i analyzes the local WFG to find *local deadlocks* in the form of a cyclic wait. If a local deadlock is detected, the agent handles this deadlock in a way that is similar to how deadlocks are resolved in a centralized system. The wait dependencies of the local deadlock are then removed upon resolution. Note that the site agents interact with the Transaction Manager of the corresponding site to detect and resolve local deadlocks. Finally, the SAg_i broad-

cast the local-deadlock-free WFG to any existing GAg.

In this study, we demonstrate the benefit of restructuring the WFG that GAg receives as input in a way that improves the overall system performance. This approach was also adopted in psychology and behavioural economics [68, 86], and recently in Multi Agent Systems [7, 119]. The idea is that the global agents believe that the WFG is local-deadlock-free and thus requires a substantially smaller CPU consumption. This technique guarantees that local deadlocks do not cause the global deadlocks. Furthermore, a site that does not contain any WFG with remote dependencies can be excluded from the deadlock detection process. This way, global deadlocks can be detected more efficiently.

Let $\{GAg_1, GAg_2, \cdots, GAg_m\}$ denote the set of m active Global Agents in our DDBS such that $2 \leqslant m \leqslant n$, where n is the number of sites. Note that the number of active GAgs at each detection interval can be managed by the current active GAgs in a joint agreement decision-making process. The mechanism for Global Agents in `ADDetect` comprises three phases as follows.

1. **Detecting the deadlocks**

   Upon receiving the local WFGs, each GAg initiates the deadlock detection process by simply merging the incoming local WFGs. Let the list of incoming local WFGs be $\{WFG_{SAg_1}, WFG_{SAg_2}, \cdots, WFG_{SAg_p}\}$, where p is the number of sites providing their local WFGs. Note that a SAg can abstain from providing its local WFG if the local WFG does not have any remote dependency by notifying GAgs through a simple message. A GAg then constructs the global WFG, $\mathcal{G}\_\mathcal{WFG}$.

   In the next step, the GAg generates the node list of the global WFG (that is,

the list of unique transactions involved in $\mathcal{G}\_\mathcal{WFG}$) as follows:

$$\mathcal{T} = \{T_i \mid T_i \text{ is a vertext of } \mathcal{G}\_\mathcal{WFG}\}$$

where the T_is are partitioned among the Global Agents. Each GAg independently searches the whole $\mathcal{G}\_\mathcal{WFG}$ to find cycles. Details of the global deadlock detection algorithm are presented in Algorithm 1.

A GAg uses Tarjan's depth-first search algorithm [144], which is the classical algorithm for finding cycles in a directed graph. When a GAg is tracing the $\mathcal{G}\_\mathcal{WFG}$, it takes the ID of the T_i as the *head edge*. The head edge must have a lower ID than a *tail edge*. For instance, if T_i is waiting for T_j, and GAg_p, $(2 \leqslant p \leqslant m \leqslant n)$, is responsible for tracking T_j in $\mathcal{G}\_\mathcal{WFG}$, GAg_p will continue tracing if the ID of T_j is greater that the ID of T_i. The only exception is when the ID of T_j is equal to the ID of T_i, and thus the GAg_p knows that a deadlock cycle has been detected. Otherwise, the GAg_p will continue searching for any existing deadlock cycle until there is no unvisited edge related to T_j. In the proposed algorithm, each deadlock cycle is detected by only one agent. Therefore, the probability of detecting phantom deadlocks[1] is reduced. The mathematical proof is provided in Appendix A.3.

2. **Resolving the deadlocks**

   In this state, when the agent detects a cycle, it tries to resolve it by a deadlock resolution algorithm. When a transaction to drop is selected, the GAg passes that transaction ID to its corresponding SAg, which will drop the transaction. The resources held by the dropped transaction are then released.

   The list of sites that are involved in that global deadlock is then created by the GAg that found the cycle. Finally, the GAg passes the resolved deadlock

[1] Phantom deadlock is a cycle detected as a deadlock which does not actually exists [109].

Algorithm 1 Deadlock Detection Algorithm

▷ On entry the following information is known

▷ $Active\_GAg \leftarrow \{GAg_i \mid GAg_i \text{ is a Global Agent}\}$

▷ $g = |Active\_GAg|$

procedure FOLLOWCYCLE_GLOBALAGENT(i) ▷ For GAg_i
  $\mathcal{G}\_\mathcal{WFG} \leftarrow$ Global *wait-for-graph*
  $\mathcal{S} \leftarrow$ {Sites that provide WFG}
  $\mathcal{T} \leftarrow \{T_i \mid T_i \text{ is a vertex of } \mathcal{G}\_\mathcal{WFG}\}$
Partition the transactions that GAg_i *is tracing*
  $AgT_i \leftarrow \{T_i \mid T_i \in \mathcal{T}, \text{ index of } T_i \bmod g = i\}$ ▷ $AgT_i \subset \mathcal{T}$
  $Deadlocks \leftarrow \emptyset$
Depth-first search
  while $AgT_i \neq \emptyset$ **do**
    $Head\_Edge \leftarrow AgT_i(1)$
    $Current\_Edge \leftarrow Head\_Edge$
    $deadlock\_Path \leftarrow \emptyset$
    foreach $T_i \in \mathcal{T}$ **do**
      $Tail\_Edge \leftarrow T_i \mid T_i \neq Current\_Edge$
      if $\exists$ wait from $Current\_Edge$ to $Tail\_Edge$ **then**
        if $ID(Head\_Edge) < ID(Tail\_Edge)$ **then**
          if $Tail\_Edge \notin deadlock\_Path$ **then**
            $deadlock\_Path \leftarrow deadlock\_Path \cup \{Tail\_Edge\}$
            REMOVE $Tail\_Edge$ from AgT_i
            if $\exists$ wait from $Tail\_Edge$ to $Head\_Edge$ **then**
              $Deadlocks \leftarrow Deadlocks \cup \{deadlock\_Path\}$
              break
            else
              $Current\_Edge \leftarrow Tail\_Edge$
    REMOVE $AgT_i(1)$ from AgT_i
Send Deadlocks to deadlock resolution algorithm
  $Involved\_Sites_i \leftarrow \{S_i \in S \mid S_i \text{ is a site of deadlocked transaction}\}$
    ▷ $Involved\_Sites_i \subseteq S$
  BROADCAST deadlocks info to $Involved\_Sites_i$

information to the corresponding SAgs.

3. **Reconsidering configurations**

When deadlock detection executes periodically, the overall performance of deadlock handling not only depends on the efficiency of the deadlock de-

tection algorithm, but also on how frequently the algorithm is executed [24, 77, 94, 113]. When transactions have a time constraint, deadlocked transactions are prone to missing their deadline unless the deadlock detection process is invoked soon enough [158]. Particularly, in a DRTDBS, the choice of deadlock detection interval presents a trade-off between deadlock detection overhead and deadlock resolution cost [75, 77, 113, 129]. Chen *et al.* proved that there exists an asymptotic optimal deadlock detection frequency that yields the minimum long-run average cost [24]. Further details are provided in Appendix B.1.

When transactions are distributed, various server sites can be involved in a deadlock cycle. Increasing the number of sites involved can influence the deadlock's persistence time and therefore raise the number of trapped transactions. In this study, we experiment with the effect of elevating the number of participating agents in the overall system performance when the number of deadlocks increases. Elder [38] discussed the potential benefits of providing more agents when resources are limited. Increasing the number of agents involved can raise the number of resolved tasks within a given time period. A collection of tasks can thereby be resolved faster with multiple agents than with individual agents. However, additional agents impose extra overhead to a deadlock detection process because the message passing between site agents and global agents increases [165].

In this study, we consider adjusting detection configuration after each detection process. To support this improvement, each GAg maintains an extra set of resolved deadlock information in the form of *memory*. Memory is embedded into each global agent's structure to keep track of the number of deadlocks in the previous detection intervals. This feature of agents gives information in advance about the past to estimate the global state of the system in

the future from individual observations. GAgs use the captured information to adjust their further actions accordingly. These actions include adjusting the detection interval and setting the number of active global agents. For example, the global agents can individually decide to reduce the interval between detection processes and employ more global agents based on whether the number of deadlocks is increasing or decreasing.

However, the autonomous decision of each agent must be communicated to other agents. We forgot to model the effect of communication overhead in this scenario. We believe that it does not measurably affect our evaluation. Change is only going to happen when there are reasonable numbers of deadlocks, in which case the WFG communication is likely much larger than the occasional agent number change message. The details of agents' memory consumption are presented in Appendix A.3.1

Figure 3.3 illustrate the execution flow representations of `ADDetect`.

3.2.3 Agent Deadlock Resolution Algorithm

In this section, first we explain the implementation details of the agents in the deadlock resolution algorithm. Then we present the deadlock resolution model and agents interaction. Finally, we formulate how bilateral distributed agreement works in distributed decision making.

Transaction Agents (TAg) are capable of resolving deadlocked cycles by cooperation and coordination with other artificial agents. Deadlock detector agents ini-

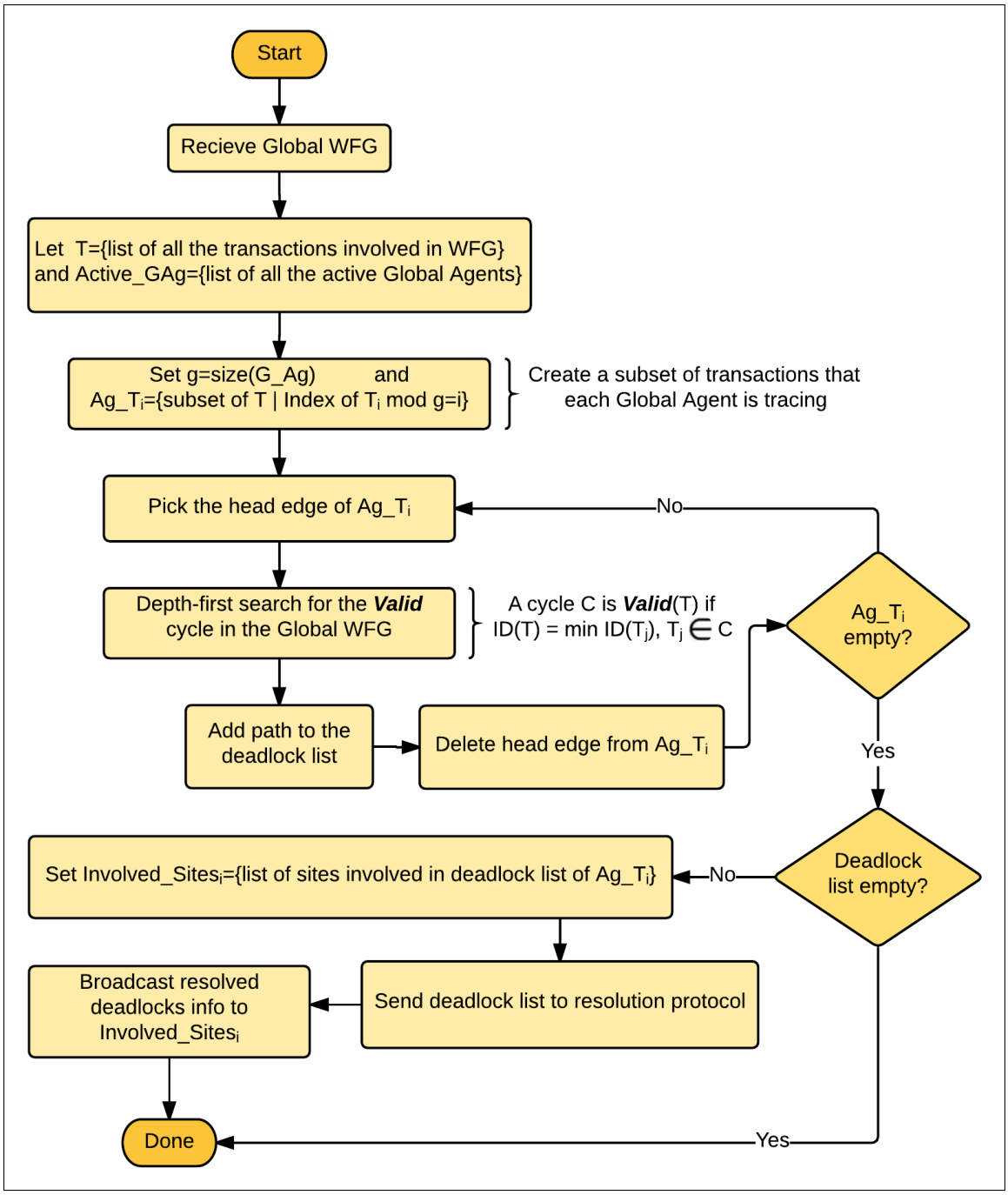


Figure 3.3: The `ADDetect` *execution flow representation. The agents participating in the algorithm execute detection flow.*

tialize and dispatch TAgs in the system. Each TAg is involved in teamwork cooperation, by broadcasting a help request or an opt-out offer in the interaction phase. Although this approach might find better transactions to drop as compared to tra-

ditional approaches (Section 2.3.3), it can also impose extra overhead. Note that multiple TAgs may reside on one site. Figure 3.4 illustrates the proposed framework of the Agent Deadlock Resolution Algorithm (ADRes) where TAg_3 decides to opt-out to break the cycle.

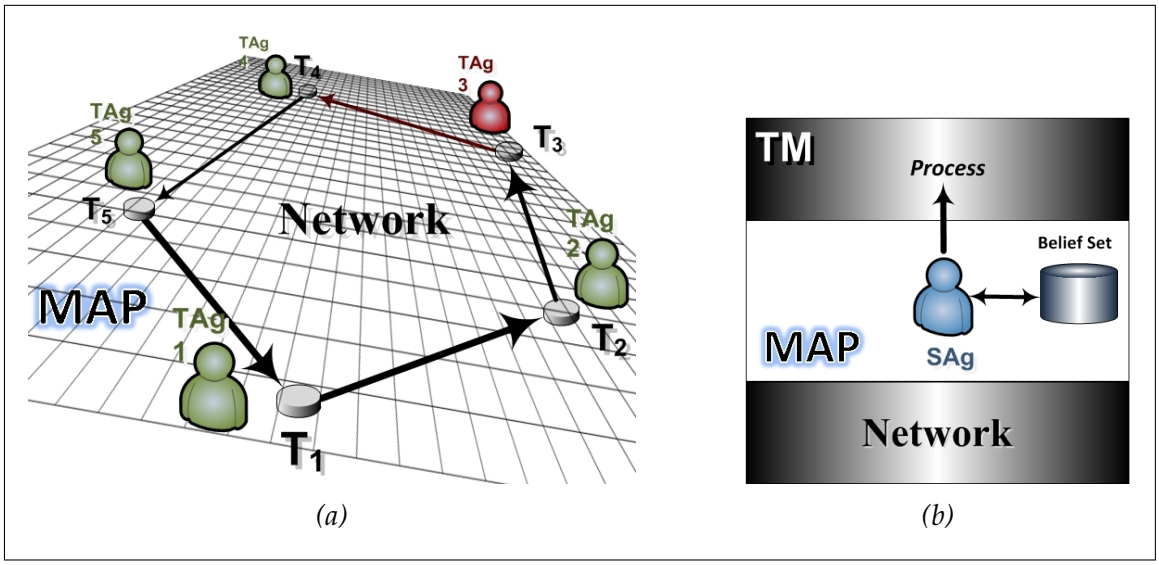


Figure 3.4: The Framework of ADRes

When a deadlock is detected, a deadlock detector agent dispatches TAgs that encapsulate the deadlock resolution strategy, together with the necessary data, and then dispatch the deadlock information to the network. The deadlock resolution algorithm embedded in the transaction agents can be adaptively altered based on the current server status and network condition (Section 3.2.4).

Generally, an increase in the number of deadlocks hampers the ADRes significantly more than the ADDetect. This is because when the number of deadlocks is relatively low, a typical TAg has enough slack time available to restart, and therefore it can make opt-out offers to other agents, but, as the number of deadlocks is increases, a TAg's ability to make opt-out offers decreases because it has less time

to restart its operations yet finish before the deadline. Hence, agents make fewer offers, resulting in fewer help acts. On the other hand, `ADDetect` is more flexible because it can adjust its teamwork to increased workload by employing more agents and reducing the detection interval.

3.2.4 Agent Deadlock Resolution Model

In the `ADRes` approach, every deadlocked transaction that is part of a cycle has a TAg that advocates for that transaction and finds a resolution. Whenever a deadlock is detected, a TAg is initiated on each deadlocked transaction. The deadlock is handled in a negotiation phase. A careful choice of victim selection is needed.

The throughput of the deadlock detection and resolution is strongly related to the amount of message passing. In this thesis, we experiment with the benefit of simple message passing in agents' *social* behaviour to reduce unnecessary overhead. Let $\{G\_TAg_1, G\_TAg_2, \cdots, G\_TAg_d\}$ denote the set of Transaction Agent groups dispatched to d detected cycles in a detection interval in a DDBS. Each G_TAg_i, $(1 \leqslant i \leqslant d)$, comprises three interaction phases: *monitoring*, *negotiation*, and *decision making*. The detailed implementation is presented as follows:

1. **Monitoring Phase**

   Let $G\_TAg_i = \{TAg_{T_1}, TAg_{T_2}, \cdots, TAg_{T_k}\}$, $(k > 1)$, denotes the set of k Transaction Agents in the cyclic wait of deadlock i. Note that the ID of each TAg is considered as the same ID of the corresponding transaction in which they advocate. Although the desire of each agent in the cycle is to meet all the deadlines of deadlocked transactions, but, satisfying timing constraint of its

own transaction is the primary intention in a deadlock resolution process. Each TAg_{T_j} ($T_1 \leqslant T_j \leqslant T_k$) checks the current state of T_j to calculate both *urgency* and *criticalness* of the transaction. Abortion should be performed on the less urgent and critical transactions only. The details of the global deadlock resolution algorithm are presented in Algorithm 2.

Algorithm 2 Agent Deadlock Resolution Algorithm for TAg_{T_i}

procedure RESOLVE
  $\mathcal{T} \leftarrow \{T_i \mid 1 \leqslant i \leqslant n\}$
  $G\_TAg \leftarrow \{TAg_{T_i}, \mid 1 \leqslant i \leqslant n,\ ID(TAg_{T_i}) = ID(T_i)\}$
  $count \leftarrow 0$
  $known(\Gamma_i) \leftarrow false$
  $no\_interest \leftarrow false$
Monitoring Phase:
  $TAg_{T_i}Params \leftarrow \{D_i, P_i, W_i, Ex_i, ST_i\}$ ▷ D_i=Deadline of T_i ▷ P_i=Priority of T_i ▷ W_i=Workload of T_i ▷ Ex_i=Execution time of T_i ▷ T_{start_i}=Start time of T_i
  $R_i \leftarrow D_i - T_{now}$
  $Urg_i \leftarrow R_i - Ex_i$
  $ExR_i \leftarrow \langle Ex_i - (T_{now} - T_{start_i}))\rangle \times 100 \times W_i \,/\, Ex_i$
  $Cri_i \leftarrow P_i \,/\, ExR_i$
  $\Delta_i \leftarrow \begin{cases} 0, & \text{if } Urg_i < 0 \\ Urg_i \,/\, Cri_i, & \text{otherwise} \end{cases}$
Negotiation Phase:
  $\Gamma_i \leftarrow \langle \Delta_i^+, -P(T_i), -W(T_i), ID(T_i)\rangle$
  $known(\Gamma_i) \leftarrow true$
  $no\_interest \leftarrow count > 0$ and $\Gamma_i < \Gamma_c$
  $count \leftarrow count + 1$
Decision Making Phase:
  See Algorithm 3 on page 53 ▷ Algorithm 2 and Algorithm 3 are concurrent

The agent TAg_{T_j} retrieves the transaction's entire execution time (Ex_j) and its assigned deadline (D_j). Note that we assume that the estimated Ex_j is accurate and known to the agent. If the agent offers to opt-out, the remaining time available to restart is defined as:

$$R_j = D_j - T_{now} \tag{3.1}$$

The agent then calculates its corresponding transaction's urgency as follows:

$$Urg_j = R_j - Ex_j \tag{3.2}$$

where a large positive value of Urg_j indicates that there is sufficient time available for the transaction to drop and restart and hopefully complete its designated tasks before its deadline; and a nearly zero value of Urg_j indicates that the remaining time for the transaction to drop and restart and successfully meet its deadline is marginal. If the remaining time is not enough for the transaction to complete its tasks (*i.e.*, $Urg_j < 0$), the agent does not have enough resources to provide help offer. Note that the value of a timed-out transaction is considered zero [109] and thus the transaction is dropped immediately.

The agent's corresponding transaction's criticalness depends on the transaction's priority (P_j) and the execution amount remaining to complete its assigned tasks (ExR_j). The priority of a transaction in the list of deadlocked transactions is a positive integer value that indicates how many transactions have a lower degree of importance than this transaction. Criticality of the transaction (Cri_j) is defined as:

$$Cri_j = \frac{P_j}{ExR_j} \tag{3.3}$$

where ExR_j is calculated as follows:

$$ExR_j = \frac{\left(Ex_j - (T_{now} - T_{start_j})\right) \times 100}{Ex_j} \times W_j \tag{3.4}$$

such that T_{start_j} denotes the transaction's start time and W_j represents transaction's workload. The more critical a transaction, the more willing its corre-

sponding agent is to stay and complete its transactions. Finally, the agent's transaction's droppability Δ_j is calculated as follows:

$$\Delta_j = \frac{\mathrm{Urg}_j}{\mathrm{Cri}_j} \tag{3.5}$$

Algorithm 3 Decision Making for TAg_{T_i}

```
▷ Algorithm 2 and Algorithm 3 are concurrent
1:  procedure RECEIVE(Δ_j)
2:      Γ_i ← ⟨Δ_j^+, −P(T_j), −W(T_j), ID(T_j)⟩
3:  lexicographic Comparison
4:      if no_interest then
5:          do nothing
6:      else
7:          if known(Γ_i) then
8:              no_interest ← Γ_i < Γ_j
9:          else
10:             Γ_c ← max(Γ_c, Γ_j)
11:     count ← count + 1
12:     if count = k then                ▷ k=Number of transactions in the deadlock
13:         if no_interest then
14:             wait for opt-out
15:         else
16:             victim ← TAg_{T_i}
17:             DROP T_i
18:             BROADCAST opt-out notification to G_TAg
19:             return
```

2. **Negotiation Phase**

   In this phase, transaction agents share their willingness to opt-out and break the cycle. While an early opt-out decision guarantees a quick deadlock resolution, an agent that drops its transaction needs to restart its tasks. Each agent that proposes a help offer must satisfy the positive urgency value condition. Otherwise, in case there is not enough time to restart, it broadcasts a help request (*i.e.*, zero droppability). All agents, including the ones who have

sent requests, receive the request or offer from other agents in the deadlock and deliberate on selecting the best option. We assume that once an agent starts the negotiation during deadlock resolution process, there is no failure in transaction execution.

3. **Decision Making Phase**

   As transaction agent j receives offers and requests from teammates, it keeps track of the maximum value (in lexicographic order) of the four-tuple $\langle \Delta_k^+, -P(T_k), -W(T_k), ID(T_k) \rangle$ so far received (Algorithm 3). When agent j has received messages from all of its teammates, it determines if it has the maximal four-tuple. If so, it selects itself as a victim, drops its transaction, and informs the global agents. It is assumed that only one agent in a deadlock can decide to drop its transaction (separate execution condition), and that transaction cannot forcibly take from an agent holding it, but can only be released by an explicit action of the agent. Note that Algorithm 2 and Algorithm 3 are concurrent. The interaction sequence for this phase is illustrated in Figure 3.5.

In this study, we also adjust the resolution decision-making process based on recent deadlock statistics. The aim is to maximize the number of completed transactions. The new combined model is called Agent Deadlock Combined Detection and Resolution Algorithm (`ADCombine`).

Figure 3.6 illustrate the execution flow representations of `ADRes`.

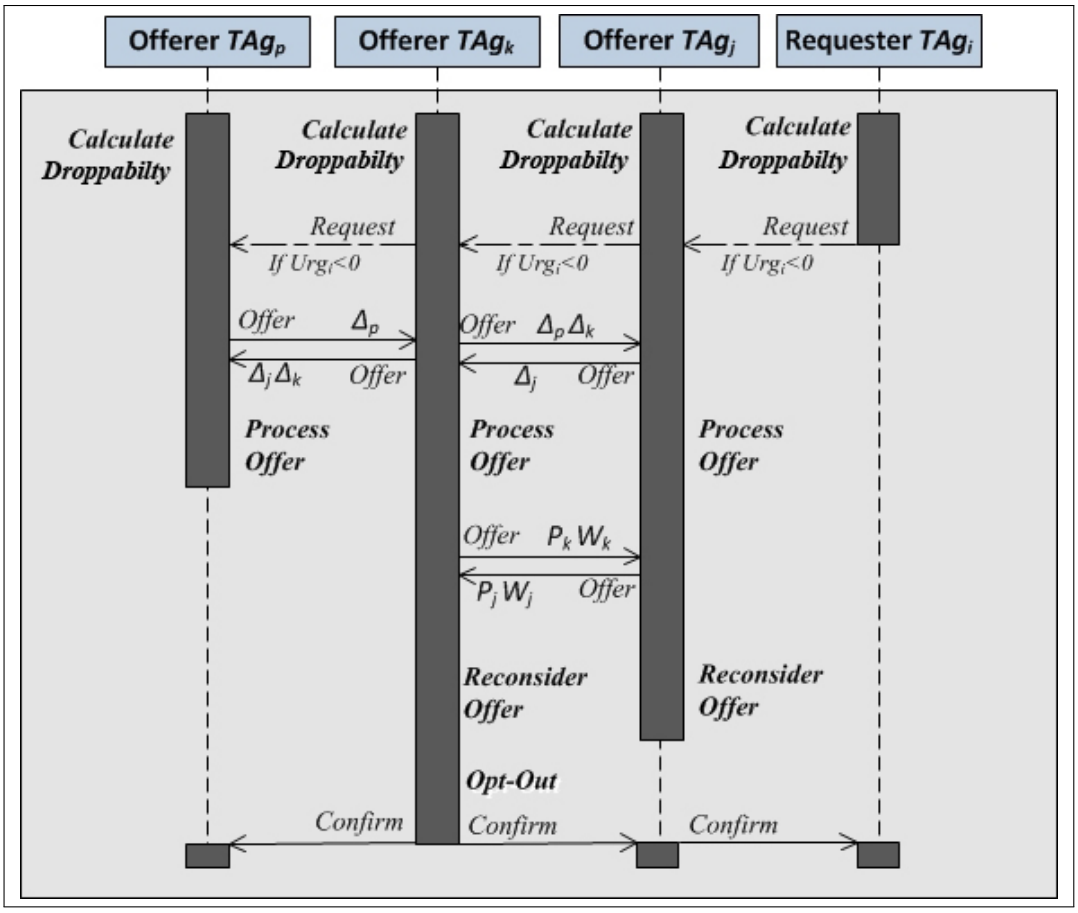


Figure 3.5: `ADRes` *Decision Making Phase*

3.2.5 `ADCombine` Algorithm

The new combined algorithm is expected to leverage the advantages of helpful agents in both detection and resolution processes. For that purpose, we need an analysis of their behaviour over the parameter space. In this development, we analyze the overhead incurred by the `ADCombine`.

In designing the combined algorithm, we seek to reduce the deadlock handling overhead, particularly in message passing. One might envision different possible combination improvements of the detector and resolver agents. One possible combined algorithm is to allow an agent to detect and resolve cycles at the same time, possibly in a combined mechanism; however, such a design increases

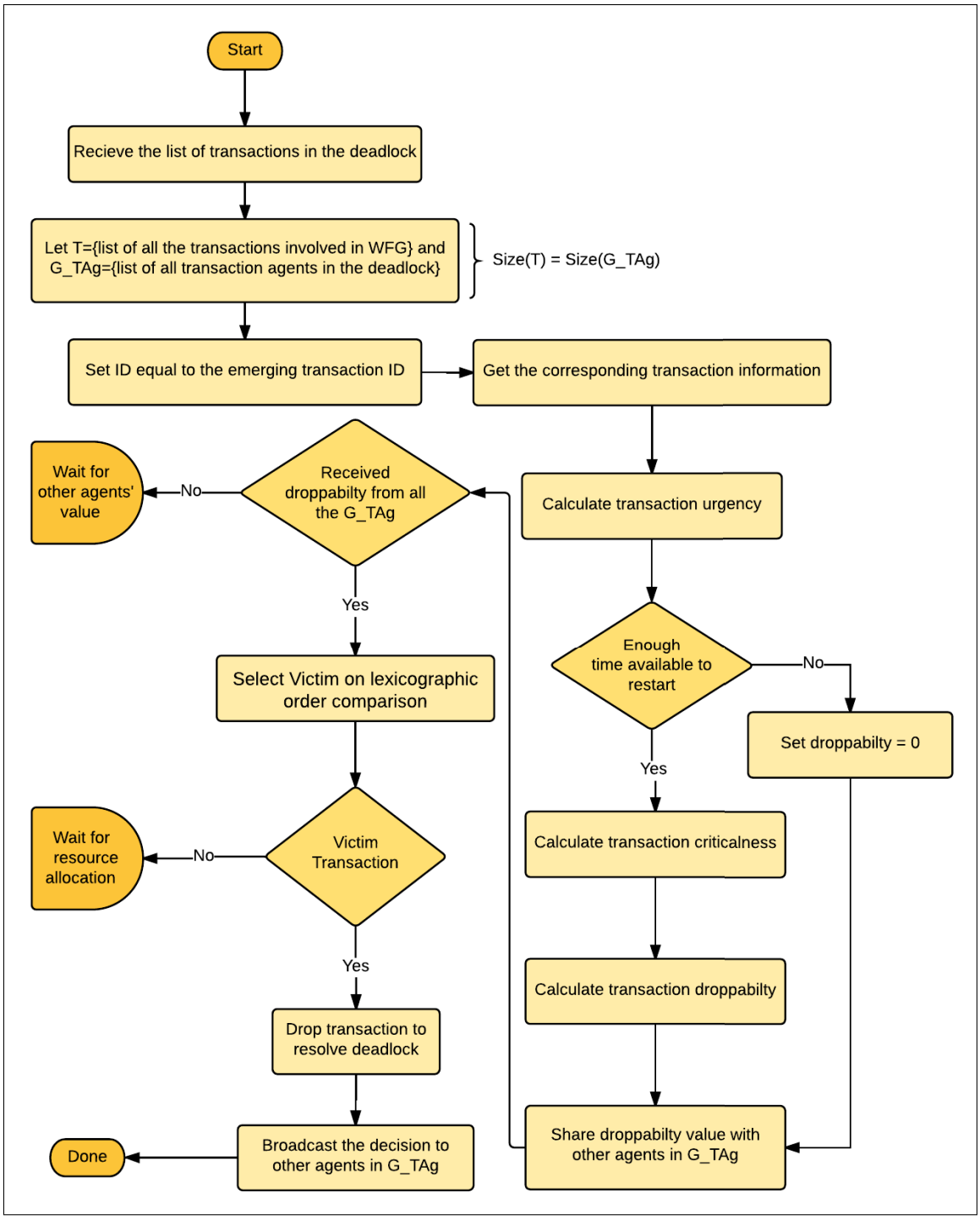


Figure 3.6: The `ADRes` *execution flow representation. The agents participating in the algorithm execute resolution flow.*

the computational cost to find the correct transaction to drop, and decreases fault tolerance.

Table 3.1: Message Complexity Comparison of Algorithms

Algorithm	Delay*	Number of Messages	Message Size
Chandy et.al [21]	$\mathcal{O}(n)$	$k \times \frac{(n-1)}{2}$	$\mathcal{O}(1)$
Obermarck [109]	$\mathcal{O}(n)$	$n \times \frac{(n-1)}{2}$	$\mathcal{O}(n)$
`MAEDD` [17]	$\mathcal{O}(n^2)$	$n \times (m+1)$	$\mathcal{O}(n)$
`ADDetect`	$\mathcal{O}(n)$	$n \times \frac{(m+1)}{2}$	$\mathcal{O}(k)$

n = the number of sites
k = the number of waiting transactions
m = the number of agents involved in the deadlock detection
* = Delay is the worst-case time complexity [120]

A design in which the interaction phase is simplified allows agents to reduce overhead. When the overhead is low, the transaction wait time decreases. An alternative design, in which the message complexity is simplified, is to let the agents first determine the transactions involved in a deadlock cycle, and then deliberate on resolving. This approach avoids unnecessary message passing; we adopt it as the basis for our combined algorithm.

The `ADCombine` utilizes a pair structure to communicate the WFG between agents, which represents the ID of each edge in the WFG, particularly in deadlock detection. Table 3.1 compares the complexity of different deadlock detection algorithms for distributed deadlocks in the DRTDBS [78]. As shown in this table, the complexity of the `ADDetect`is close to or better than the existing algorithms in the worst case scenario. Details are provided in Appendix A.3.

The experiment setup, results, and interpretations are provided in Chapter 5.

Chapter 4

System Model and Simulator Architecture

This chapter presents the methodology deployed in this research. The first section describes the design and architecture of the distributed real-time transaction processing simulator, which is considered as the research tool to analyze deadlock handling algorithms in a Distributed Real-Time Database System (DRTDBS). The second section provides the performance analysis as well as the considerations required in the design of the agent based algorithms to handle distributed deadlocks in a real-time environment.

4.1 The Simulated System

Studying and developing a model to provide insights into the selection of appropriate approach is possible through simulation. The simulation model represents the essential portion of daily life, for example, distributed databases in this research. Experimenting and conducting research directly on an actual DDBS is not reasonable due to limitations such as cost, time, complexity and error-prone nature. Furthermore, the empirical results of the previous works are not available on a real system. Thus, using a simulation provides a better environment in which to analyze the models in different scenarios.

Simulations of a system can be either discrete or continuous [162]. A continuous event simulation refers to the continuous changes in the state of a system over a time period. A space rocket is an example of continuous event simulation where its state, such as location and speed, is changing continuously over time. When the state of a system changes based on the occurrence of events at discrete points in time, it is called discrete event simulation. For example, in a medical clinic, the number of patients in the queue can be considered one of the states. The events are patient arrivals and patient departures. Both discrete and continuous event simulations require mathematical probability distributions in order to randomize the components and thus provides a more realistic simulation. We developed a discrete event simulator in this study. The details of the simulator events are provided in Section 4.4.2.1.

In a distributed real-time database model, each site operates in a similar way to a centralized system. these sites consist of several internal modules such as a Transaction Manager, Data Manager, and Lock Manager [12]. The additional

module, Transaction Generator, presented in Section 4.4.2.2. In the following sections, the details of each internal module are discussed. Figure 4.1 represents the high-level view of these modules.

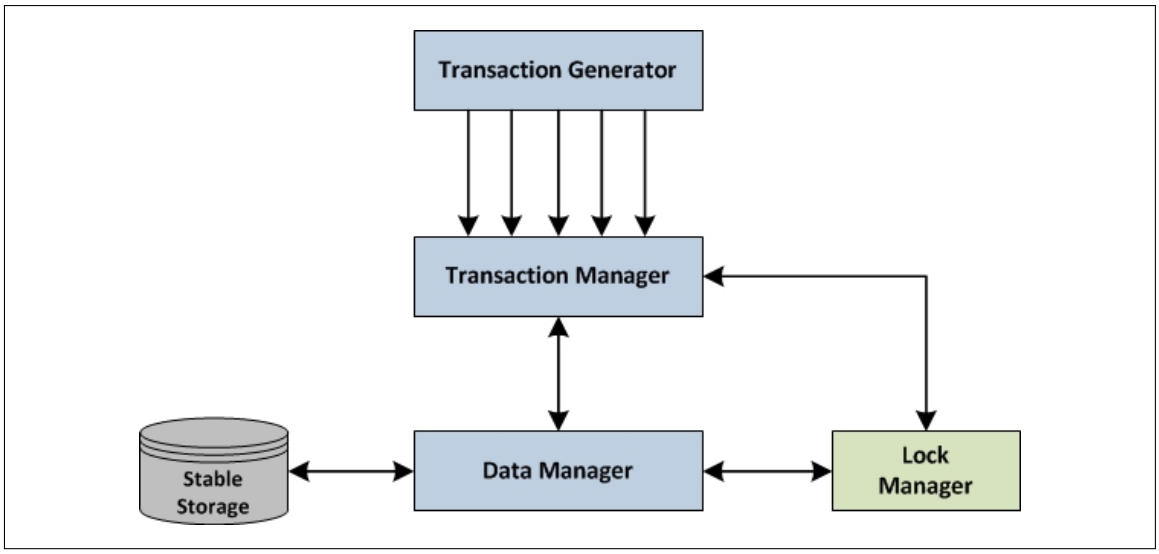


Figure 4.1: Internal and External Modules of a Database System

4.1.1 Transaction Manager

In a transaction processing environment, the Transaction Manager (TM) is the core component [5]. The Transaction Manager is a component that manages the concurrent execution of transactions by employing a Concurrency Control Protocol (CCP). The serialization of transaction execution is ensured by the TM. When the TM accepts a transaction operation from the Transaction Generator, it performs one of the following based on the order of each transaction execution:

Executes the operation by passing it to the Data Manager (DM) and receives the results consequently. The results are then committed to the Data Manager by

the Transaction Manager.

Delays the operation by holding the transaction in a queue within the Transaction Manager, and it will be processed with either execution or rejection at a later time.

The TM in a DDBS is responsible for database consistency. Furthermore, in a real-time environment it must also consider the time constraint on transactions as well [23]. The TM sequence diagram of our simulator is illustrated in Figure 4.2.

4.1.2 Data Manager

The DM at each site receives transactions' operations from the TM once the order of execution is determined. Typically, in a database system, the pages of data are permanently stored in *stable storage* [83]. When a transaction requests access a page, the DM to locates the page in stable storage to make it available for that transaction. Transactions' operations are then processed by using one of the read, write, commit, or abort actions in the DM. When a transaction is performing a read and write operation, the TM submits disk jobs to DM to perform the operation against the database. However, when a transaction commits or aborts, an update or rollback action is performed respectively by the DM using CCP to guarantee the consistency of the database (Section 2.1.4).

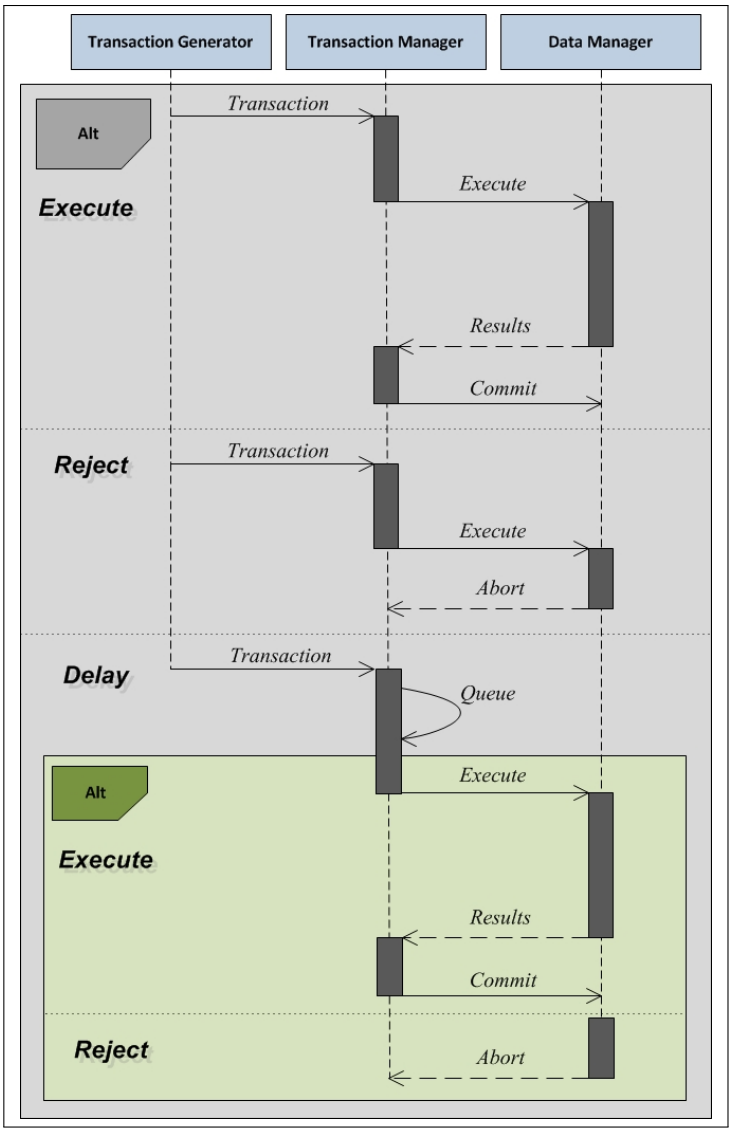


Figure 4.2: Transaction Manager Sequence

4.1.3 Lock Manager

In a DRTDBS, every transaction entering the system has some associated properties such as arrival time, deadline, priority, and so on [8]. A Lock Manager is a component responsible for ensuring isolation between concurrent transactions. The Lock Manager maintains at most one exclusive lock or a finite number of shared locks on each page at any given time. Whenever a lock request for a page is re-

ceived by a Lock Manager, the Lock Manager locates the page in the database. The lock request can either be granted or be blocked, based on the status of the existing lock on the requested page. The life-cycle of a transaction from being generated to completion of its tasks is demonstrated in Figure 4.3.

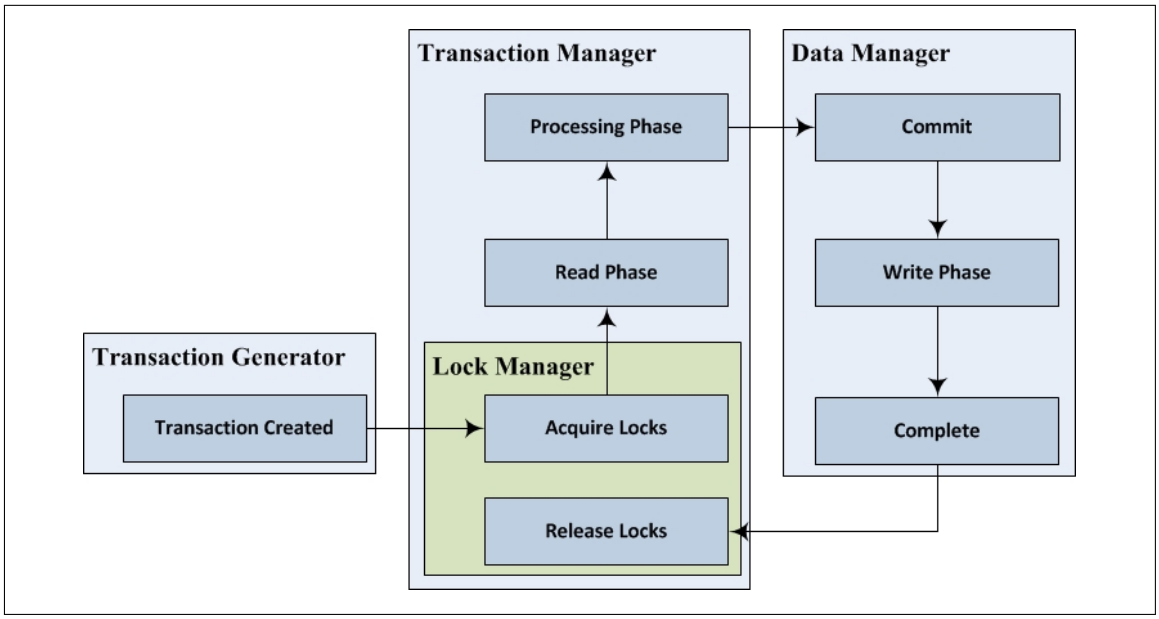


Figure 4.3: Transaction Life-Cycle

4.2 Deadlock Handling in The Simulated System

In the simulated system, deadlock detection and resolution algorithms are adopted to handle deadlocks. These algorithms are equally shared between all sites and are initiated within a time interval. Different deadlock detection approaches are implemented in this simulator.

After a deadlock is detected, information regarding the deadlocked transaction is required to determine a resolution. The deadlock detection algorithm

generates a list of transactions that are involved in the deadlock and passes it to the deadlock resolution algorithm. The deadlock resolution algorithm then determines which transaction to abort to resolve the deadlock. The deadlock resolution algorithms implemented in this simulator are as follows:

First Deadlock Resolution: The transaction that appears at the top of the deadlock list is selected as the victim transaction to abort. This algorithm is also known as random selection.

Priority Deadlock Resolution: The transaction with the lowest priority in the deadlock list is selected as the victim transaction to abort.

Agent Deadlock Resolution Algorithm: The transaction with the lowest droppability in the deadlock list is selected as the victim transaction to abort (explained in Section 3.2.3).

4.3 The Simulated System in This Study

In a distributed database model, databases are scattered across physical locations called sites. Each distributed database is modeled as a collection of pages that are uniformly distributed across all the sites. Each page is replicated once, and that each server has a contiguous range of pages, and that the page numbers used by transactions are uniformly randomly generated. Sites are connected through a network. The details of the network architecture and node configurations are provided in the following.

4.3.1 Network Architecture

A network topology denotes the physical arrangement of a network, including its nodes and connecting lines. In our case, sites and pipes are nodes and connecting lines respectively. The simulated network topology consists of multiple sites with one node within each site. It is important to note that each site is a Real-Time Database System (RTDBS). In general, nodes are connected through a Local Area Network (LAN) within a site, and sites are linked by a Wide Area Network (WAN) as shown in Figure 4.4. In our simulator, sites are connected by a WAN.

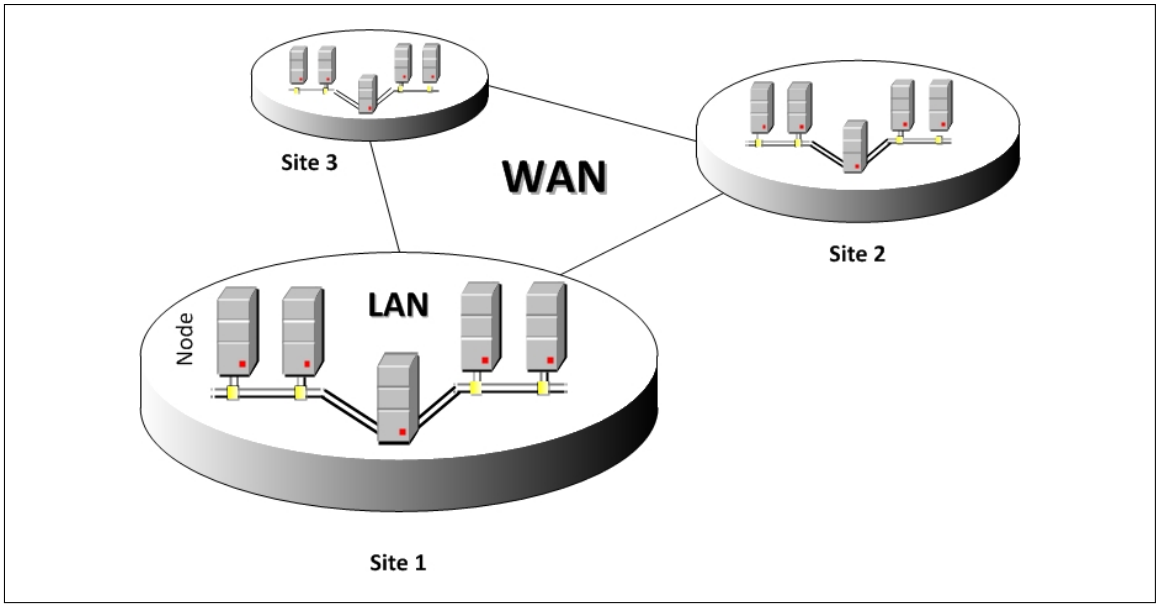


Figure 4.4: Network Architecture

This simulator uses a hypercube as the network topology. The hypercube topology is a type of network topology used to connect multiple processors. It has been commercially applied in numerous scientific applications [2, 44, 132]. The communication pattern for a hypercube network is illustrated in Figure 4.5.

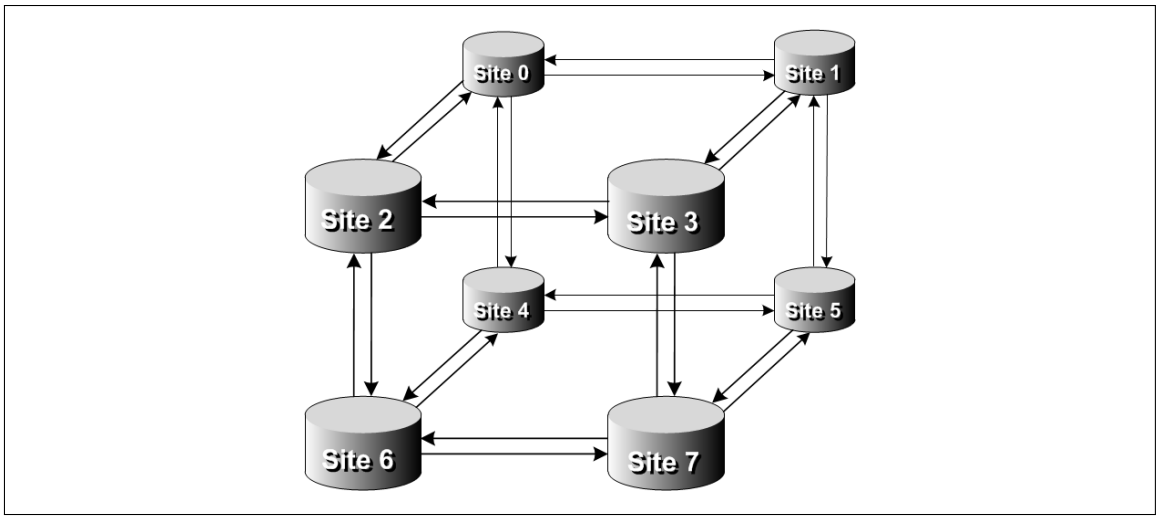


Figure 4.5: Communication Pattern for A Hypercube

Each of these network connections is a pipe that denotes a bi-directional connection between one site and another. A router at each site is responsible for all the network connections between itself and other sites. Additionally, each network node stores its own routing table that consists of all the possible destination nodes. A message might need to travel through multiple intermediate sites to reach its final destination.

Each message may be a complex data structure. In the course of a simulation run, messages are scheduled to occur, or they may be exchanged or even canceled. A message holds two attributes: A *source* node which is the site from which a message initiates, and the site to which a message should arrive at known as the *destination*. Every network connection contains the following parameters:

- *Bandwidth* is the maximum number of message units that can be carried from one point to another in each tick. The number of events carried by a message is message unit.
- *Latency* is the amount of time taken by a message to travel from its source to

the desired destination server through a network connection.

4.3.2 Node Architecture and Configuration

A node within the simulator is characterized by many components such as processor, disk, and local transaction generator. These components are described in the following paragraphs.

The processor organizes and controls all processes in a node. A node is equipped with one processor, and a processor can deal with at most one data item at a time. The processors are characterized by one parameter, *Processing time*, which is the amount of time taken by a processor to process one page.

The disk is responsible for organizing and managing all the pages in a node. Disks are non-volatile storage devices in which a specified set of pages is stored. A disk is capable of performing only a one-page read and write operation at a time. Pages can be replicated in multiple nodes. However, pages that are replicated across multiple nodes require extra configuration on CCP. The disks have two parameters: *Access time* which is the amount of time taken by a disk to read from or write to a page, and the number of pages that are stored on each disk known as *page range*.

The final component of the node is the local transaction generator which is responsible for creating transactions to be submitted at that node. The properties of transactions are defined in the transaction generator's attributes which are defined as follows:

- *Size* is the total number of transactions to be created during a simulation run.
- *Arrival Interval* defines the mean inter-arrival time between the two transactions arriving at a node.
- *Slack Time* represents the extra time allocated to a transaction to complete its operations and still meets its deadline.
- *Work Size* is the maximum number of accessible pages for a transaction during its execution.
- *Update Percent* defines the probability of a page being written to or updated by a transaction as a function of the total number of pages access.

Node architecture is partly described by the components listed above. However, the basis of transaction processing in a node requires additional characteristics. The system employs the use of a *transaction timeout* attribute. In the latter case, the system employs a timer that is activated upon initiation of each transaction and when the timer expires the transaction is aborted. *Maximum active transactions* is another node attribute which is the highest number of transactions that are allowed to run concurrently on a node. In the case of exceeding that limit, the newly arriving transactions are held in the *transaction wait queue*.

When a transaction is waiting in a queue, a priority protocol determines which transaction within a priority queue should be processed next. In this simulator, a priority protocol controls the behaviour of all of the priority queues within a node, including the transaction queue. Prioritizing protocols implemented in the simulator are:

Earliest Deadline First: The priority of a transaction is determined according to the deadline. The transaction with the earliest deadline is considered as the

highest priority transaction in order to meet its deadline.

First Come First Serve: The priority of a transaction is assigned based on the order of its arrival and is served in a First-In-First-Out (FIFO) queue.

Least Slack Time First: The priority of a transaction is allocated based on the transaction's total amount of time provided as a slack time to complete. The shortest slack time gets the highest priority.

Random Priority: The priority of a transaction is allocated randomly.

CCPs are used to guarantee the isolation property of transactions for both committed and aborted transactions [12]. Each transaction's request for a lock on a page is handled through these protocols. A shared lock on a page is granted to transactions requesting reading access, and an exclusive lock is given to transactions requesting write operation on a page. Furthermore, locking algorithms based on data access patterns of operations can be classified into static or dynamic locking [101]. In static locking algorithms, a transaction requests locks on all the pages that will be required during the transaction's lifetime, before the initiation of its execution. Locks will be released when the transaction is completed or terminated. One of the most well-known static locking algorithms is greedy locking [163] which supports the locks on replicated pages across nodes as well. However, in dynamic locking algorithms, a transaction will request for access to data items as the need arises [8]. In general, dynamic locking algorithms proved to perform better than static locking algorithms [8, 63, 150]. In this research, we use Dynamic Two-Phase Locking (`D2PL`) [60] as our Concurrency Control Protocol. In this protocol, a transaction requires a page to be locked before it starts processing that page.

4.4 The Simulator

This section describes the adopted DRTDBS model followed by the design and architecture of the distributed real-time transaction processing simulator.

4.4.1 Distributed Real-Time Database System Model

In this research, we used the simulation model for DRTDBS, proposed by Ulusoy and Belford [153]. The model can be used in evaluating various components of transaction scheduling algorithms, including deadlock detection and resolution algorithms. The model supports modeling the relevant performance behaviour of each component. Therefore, the users can study the system under diverse real-time database environments to get insights about different approaches. Two performance metrics used in the evaluations are the percentage of transactions completed on time, success ratio, and the average of incurred overhead [153].

The adopted model simulates the characteristics of a Distributed Database System (DDBS) in which transactions are associated with timing constraints. Let $\{S_1, S_2, \cdots, S_n\}$ denote the set of n sites in our DDBS. We experiment with eight sites, and it is assumed that the database at each site has the same size. The replicated data distribution structure denotes that the site S_k contains data item D, which can have 0 to $n-1$ remote copies at the other sites (explained in Section 2.1.1). Let $N(D)$ indicate the number of replicas of D, including the original. We assume a double replication of each page in the simulated system.

Each distributed database is modeled as a collection of pages that are uniformly distributed across all the sites. The transactions' consistency in the distributed system is guaranteed using the "*read-one, write-all*" approach. If a local copy of the data exists on the originating site, the read operation can be performed; otherwise, any remote copy of the data is accessed with uniform probability. However, writing on a data requires all the copies of the data to perform a "*write-all*" operation.

The performance models of RTDBS are not sufficiently developed to evaluate real-time capabilities of the system [152]. The database community has developed several benchmarks, TPC-B [28] and TPC-C [29],to evaluate the performance of traditional transaction processing databases. However, the benchmarks for real-time systems such as [90] are focused too much on disk performance. Additionally, the existing benchmarks do not consider adequate properties of database transaction management. Therefore, developing performance model and benchmarks to evaluate real-time functions are needed for DRTDBS.

The number of transactions that arrive in a time interval have an *Exponential* distribution [135]. Each transaction has an associated firm time constraint in the form of a deadline. Each transaction is executed to complete its assigned tasks, even if it misses its deadline. The deadline (an instant in time) is computed using the following formula:

$$T_{deadline} = T_{arrival} + T_{process} + T_{slack} \tag{4.1}$$

The processing time (T_{slack}) is the total time that the transaction requires for its execution. The slack factor is a constant that provides control over the slackness of

the transaction deadlines. In this model, each transaction supports the single master, multiple cohorts structure. The cohorts are dynamically created as needed. At each site, there can be at most one operating cohort of a transaction. For each executed transaction, the TM is responsible for identifying which server stores copies of data used by transaction. The data item is referenced by the transaction using the global data dictionary available at each site. Then the cohort(s) of the transaction at the relevant sites are activated to perform the operation on one or more data items. However, the master transaction does not perform any database operations before the coordination of cohort transactions. The atomic commitment of each transaction is provided by the Two-Phase Commit (2PC) protocol (explained in Section 2.1.6) [55].

The DM is responsible for providing IO and CPU services for reading and writing data and processing data respectively. Each server is assumed to have one CPU and one disk. A summary of the simulation configuration and workload parameters is listed in Table 4.1. Also, Table 4.2 determines the default values of the system transactions parameters used in the simulation experiment. All sites of the system are assumed to be running under identical simulation configuration. We used a fixed value for IO and CPU services. The detailed calculation of IO and CPU consumption formulated in Appendix A.1 and Appendix A.2.

4.4.2 Additional Modules

The discrete event simulator was developed to experiment with deadlock handling in a distributed real-time transaction-based database system. It provides various parameters for the user to configure the system for different scenarios. Various

Table 4.1: Simulation Configuration Parameters and Values

Parameter	Description	Value
NumSites	Number of sites in the database	8
NumPages	Number of pages in the database	Varies
MaxActiveTrans	Maximum number of simultaneous transactions execute on a site	Varies
Topology	Network arrangement of the servers	Hyper Cube
DetectionInterval	The time between deadlock detections	Varies
CPUTime	Data item process time	15 ticks
IOTime	Disk page access time	35 ticks
MessageProcess	Communication message process time	2 ticks
BandWidth	Maximum message size that can be sent down a network for each tick	1000 message unit/tick
Latency	Communication message delay	5 ticks
DataDistribution	Replication protocol	Partial Replication
CCP	Concurrency control protocol	D2PL
PP	Priority protocol	Earliest Deadline First
DDA	Deadlock detection algorithm	Varies
DRA	Deadlock resolution algorithm	Varies

modules and algorithms such as deadlock detection and resolution algorithms can be added to the simulator to experiment on analyzing their performance. It also supports the considerations required in the design of the agent-based algorithms to handle distributed deadlocks in a real-time environment. The modules communicate by exchanging messages. In effect, the analyst can easily substitute a module instance with a different instance of the same type. It adopts the object-oriented de-

Table 4.2: Simulation Transaction Parameters and Values

Parameter	Description	Value
UpdateRate	Transaction page update probability	Varies
ArrivalInterval	Mean interval time of transactions at a site	Varies
WorkSize	Number of pages required by a transaction	2-10 pages
SlackRate	Average ratio of slack time of a transaction to its execution time	2
TransPerSite	Mean number of data items accessed by each transaction	300
TransTimeout	The waiting time of a transaction before timeout	5000 ticks

sign of Java™ [52] and this allows flexible extension of the currently implemented modules. The components of the discrete event simulator include entities, a simulation clock (tick[1]), and event handlers such as event queue, an event scheduler, and event processor. The additional modules of the simulator are explained in the following sections.

4.4.2.1 Simulator Events

In the DRTDBS simulator, events are processed sequentially. Each event is initiated by entities and inserted into the event queue based on their execution time. Events at the top of event queue get extracted by event scheduler and passed to event processor. Basically, events in the simulator are considered as actions performed by various components such as message passing between two sites, a page processed by a transaction, a transaction requesting a data item, and so on. [139]. Once the event is fully processed, the state of the system gets updated. Event queue will

[1]A tick is a time unit used to measure a discrete amount of time in which one or more events can be executed

process every event added to the queue until the end has been reached.

4.4.2.2 Transaction Generator

Essentially, the interaction between transactions and the database in a database system is possible through the Transaction Generator (TG). The TG is a component of the simulator to imitate the incoming transactions in the real-world. The TG is engaged in generating transactions and assigning priorities to transactions by using a specific priority assignment protocol (Section 2.1.3). In a real-time environment, each transaction is associated with a time constraint deadline as well. The TG's main responsibility is passing the created transactions to the TM. Particularly, in a distributed database environment, the TG is responsible for ascertaining the destination site for the generated transactions and thus transmitting that transaction to the appropriate Transaction Manager of a different site (Figure 4.6).

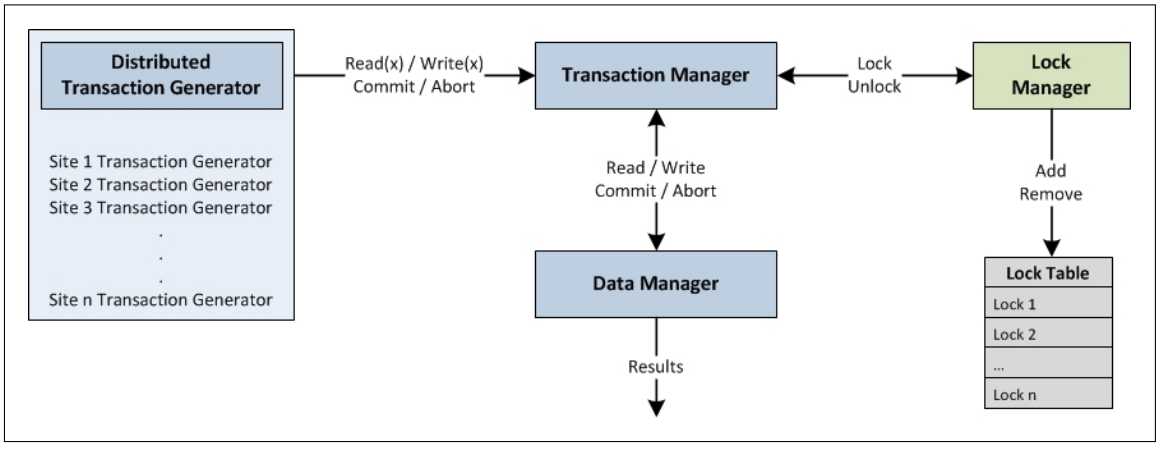


Figure 4.6: The Interaction Between Internal and External Modules

4.4.3 Simplifications

This section describes the changes made to the simulated system to simplify the simulator:

- To model agent interaction protocols without explicit synchronization details, it is assumed that both Site Agent (SAg) and Global Agent (GAg) perform actions synchronously in each *detection interval*.
- From the way Multi Agent System (MAS) modeled, we presumed that agents communicate only at the start of each detection interval, in a sequence of asynchronous *phases*.
- During the agent's negotiations, we assume that there is no failure in the transaction execution.
- The transactions global information is assumed to be known to each agent.
- The network connection is assumed to be reliable having the following characteristics:
  - No message is lost in the system.
  - Each message is delivered within a finite amount of time.
  - The network does not get partitioned[2].

[2] A network partition of a system refers to the failure of a network connection that causes a network to be split into multiple sub-systems.

4.4.4 Simulation Set-up

The adopted simulator implements appropriate abstractions which allows one to reuse existing functionality for the development of new algorithms and operate with already implemented ones. Moreover, there is the possibility for a user to alter the basic implementation of transactions with a different perception of serializability such as nested transactions or advanced transaction models [39, 46, 51].

The probabilistic consumed-time model is implemented in the simulator in order to imitate a more realistic model of time consumption for CPU and IO when processing a transaction in the DDBS. Expected CPU and IO utilizations are modeled as system parameters, allowing users to study resource utilization under varying scenarios. The details are formulated in Appendix A.

4.4.5 Architecture Considerations

In designing and developing the simulator architecture, we have also considered the possibility of modeling complex protocols. We used Java™ [52] as our base implementation language due to its object-oriented design and for its flexible cross-platform environment. All the developed protocols within the simulator adopt a common fundamental framework. This framework provides the groundwork for implementing modular algorithms such as the deadlock detection algorithms, deadlock detection algorithms, and priority assignment protocols. Each deadlock detection algorithm and deadlock resolution algorithm are pluggable components of a server. Exploiting hierarchical class structure provides several categories of

deadlock detection and resolution algorithm. These categories include a deadlock detection algorithm baseline; deadlock detection algorithms which inherit from the baseline algorithms and use Wait-For-Graph (WFG) for deadlock detection; and agent-based deadlock detection algorithms which inherit from WFG protocols. Simulation categories in deadlock resolution algorithms consist of deadlock resolution algorithm baseline and agent-based deadlock resolution algorithms which inherit from deadlock resolution algorithm baseline. The simulator consists of over ten thousand lines of code and eleven libraries for different components and protocols. The simulation UML diagram is presented in Figure 4.7. More detailed diagrams are presented in Appendix A.4.

One of the key components in our design consideration was an efficient event handling model capable of conducting millions of events in a simulation run. A discrete event model is adopted due to the behaviour of transaction processing system [111]. A priority queue is used to manage the events in the simulation. The events are added to the queue with their simulation time used as the priority. Note that each event is added to the queue with its simulation time. The execution of the simulation proceeds by repeatedly pulling from the top of the queue and executing the event thereon. A new event is created using the following method:

```
private PriorityQueue<Event> queue= new PriorityQueue<>(this::compare);

public int compare(Event e1, Event e2) {
        // Compares with simulation time
       return e1.getTime() - e2.getTime();
}

public void addEvent(Event e) {
    // Adds the event in the event queue
    // In a chronological order
   queue.add(e);
}
```

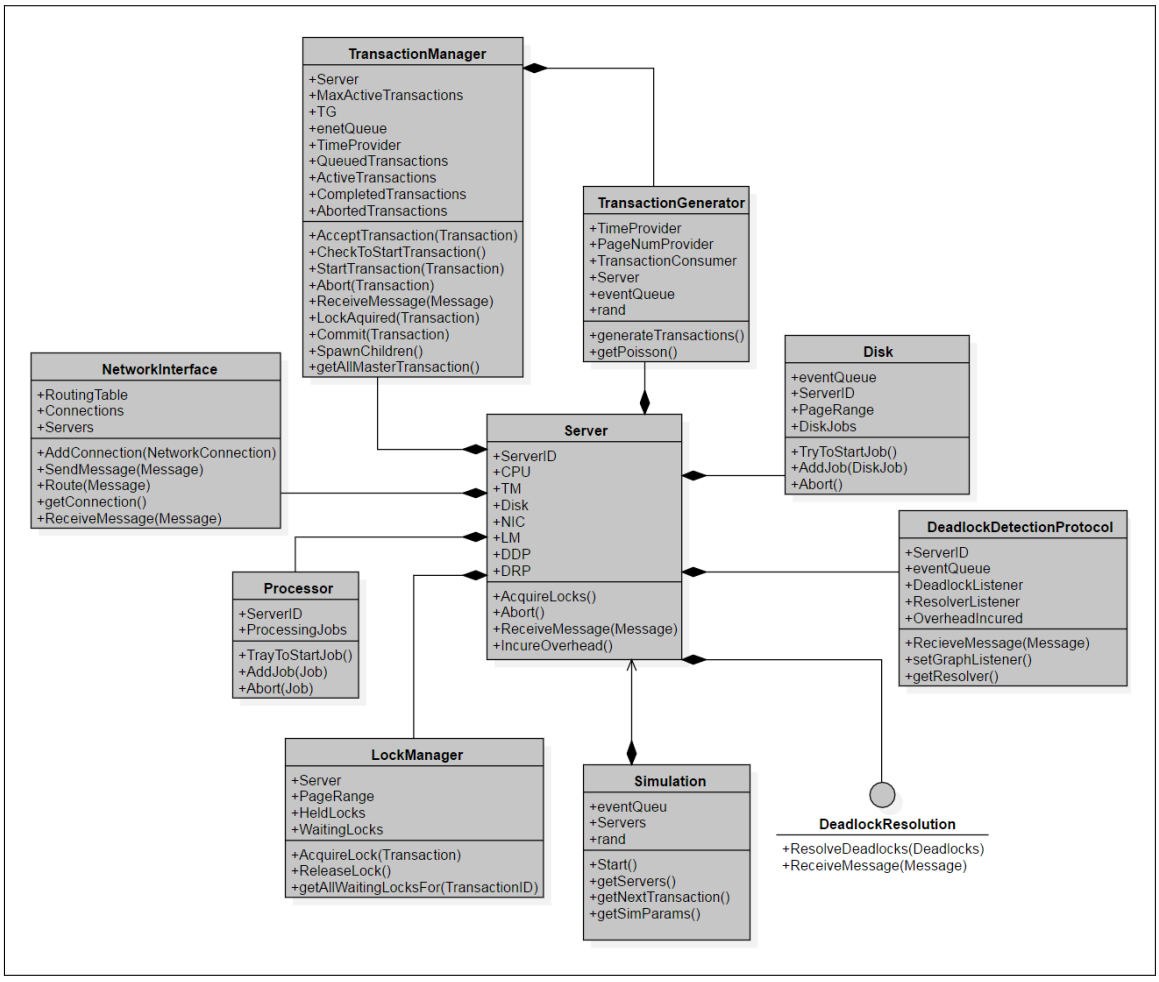


Figure 4.7: Simulation UML Class Diagram

The newly created event object holds the name of the runnable job within the object and is stored in the global event queue. The event is processed through the event queue with the following methods:

```
Runnable job = e.getJob(); // Get the name of the runnable job in the
    object
job.run();                 // Invoke the execution
```

The statistics class is developed to analyze the performance of the algorithm and keep track of the key events. The statistics accumulate the information regarding the transactions executed time and whether the deadline is met. Moreover, it collects the deadlock handling data, including the number of cycles found and resolved by the deadlock detection and deadlock resolution algorithm. The gathered information is represented by graphs that are displayed in real-time within the running simulation (Figure 4.8). The ability to track the generated WFGs while the simulation is executing is also available.

The method for collecting a completed transaction information is:

```
Statistics stats = new Statistics();

private void complete(Transaction t) {
        LockManager lm = server.getLM();
        t.getReadPageNums().forEach(pageNum -> lm.releaseLocks(t.getID(), pageNum,
            t.getDeadline()));
        t.getWritePageNums().forEach(pageNum -> lm.releaseLocks(t.getID(),
            pageNum, t.getDeadline()));

        int time = getTime();
        boolean completedOnTime = t.getDeadline() >= time;
```

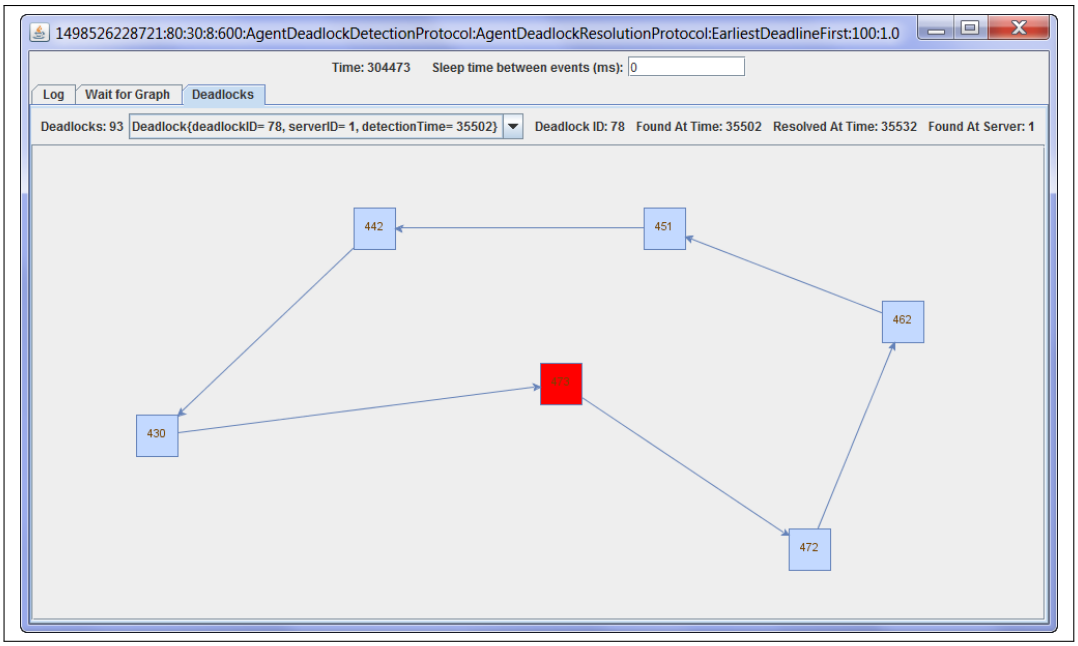


Figure 4.8: A detected deadlock with the corresponding information. Each square represents a transaction in the cyclic wait. In this example, the transaction #473 is selected as the victim

```
// Set the transaction status as completed
t.setCompleted(true);

// Log the completed time
t.setCompletedTime(time);

if (!(t instanceof CohortTransaction)) {
        // If completed its tasks before its assigned deadline
        // Then add it to transactions completed on-time list
        if (completedOnTime)
                stats.addCompletedOnTime(t.getID());
        // Else add it to transactions completed late list
        else
                stats.addCompletedLate(t.getID());
}

// Add to the list of completed transactions and remove from active
    transactions list
completedTransactions.add(t);
activeTransactions.remove(t);
```

```
        // Integrity Check
        lm.getWaitingLocks().values().forEach(locksLists -> {
                locksLists.forEach(lock -> {
                        if (lock.getTransID() == t.getID())
                                throw new Exception(serverID + ": Transaction " +
                                    t.getID() + " just completed but it has waiting
                                    locks still! (Page " + lock.getPageNum() + ")");
                });
        });

        lm.getHeldLocks().values().forEach(locksLists -> {
                locksLists.forEach(lock -> {
                        if (lock.getTransID() == t.getID())
                                throw new Exception(serverID + ": Transaction " +
                                    t.getID() + " just completed but it has held
                                    locks still! (Page " + lock.getPageNum() + ")");
                });
        });
}
```

The primary performance metric of the experiments is the ratio of completed transactions before their deadline expires to the total number of input transactions known as *Percentage Completed On Time (PCOT)*. The study analyzes the performance of the deadlock detection and resolution algorithms under different workloads by changing the arrival interval of the transaction, the number of pages available for each site, the update ratio of the transactions, and so on. Furthermore, transactions in a DDBS have to acquire the needed data from different nodes that geographically span in a distributed area in order to complete the required process. Thus, the communication overhead can have a significant effect on the system throughput [24]. To test a more realistic situation, network communication *overhead* was considered the second performance metric. Note that the overhead has different interpretations in a different context. However, a combination of message

passing overhead and WFG traversing overhead is considered as the communication overhead in this study.

Finally, MySQL™ database [64] was used to store our collected results based on its structured query language and for its flexible cross-platform environment. Once the simulation completes, these statistics can be viewed using the MySQL™ Workbench [104]. MySQL™ Workbench, developed by MySQL AB® [64], is the official integrated environment for MySQL™ to manage the database design and modeling. An example of MySQL™ Workbench environment is illustrated in Figure 4.9.

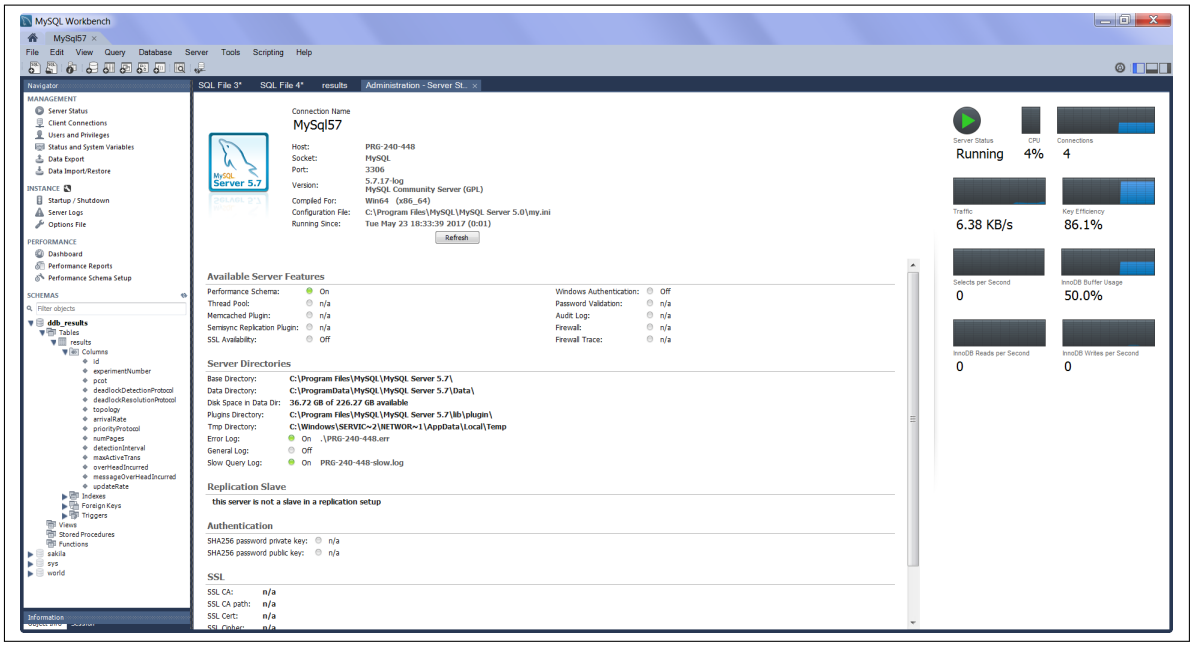


Figure 4.9: MySQL™ Workbench Environment

The accumulated data is used to analyze the performance and the efficiency of adopted deadlock detection and deadlock resolution algorithms in a DRTDBS. The application designed to visualize the collected data in the simulator is called the *ResultViewer*. These data are displayed by various graphs using the ResultViewer with proper metrics in order to evaluate different protocols by quantifying their

collective effects in a meaningful manner.

Chapter 5

Experimental Results and Evaluation

This chapter presents the performance results for using agent teams that employ our new deadlock detection and resolution algorithms, in comparison with two previously existing algorithms. We provide simulation results for three distributed deadlock detection and resolution algorithms, using the model proposed by Ulusoy *et al.* [153]. The results reveal some of the most influential performance and efficiency trade-offs to be taken into account when a distributed deadlock detection and resolution algorithm has to fulfill a given set of performance design goals.

5.1 Performance Comparisons

The following simulation experiments, the performance compare of: (a) Agent Deadlock Detection Algorithm (`ADDetect`), (b) Mobile Agent Enabled Deadlock Detection (`MAEDD`), (c) and Chandy *et al.* algorithms. The metric used is the per-

Table 5.1: Simulation Baseline Parameter Settings for Deadlock Detection Algorithm

Parameter	**Value**
NumSites	8
NumPages	80
MaxActiveTrans	30
DetectionInterval	100 ticks
BandWidth	1000 message unit/tick
Latency	5 ticks
UpdateRate	100 %
ArrivateInterval	600 ticks
Topology	Hyper Cube
PP	Earliest Deadline First
Work-Size	2-10 pages
TransPerSite	300
DRP	Priority Deadlock Resolution

centage of transactions completing on time (PCOT). The deadlock handling cost (communication overhead) is used as the secondary metric. In the experiments, the baseline parameters are selected as shown in Tables 5.1, and a selected parameter is varied to compare performance of the three algorithms over a range of circumstances. The input range for each parameter is selected in a way that it shows noticeable results.

The accumulated results are also considered with the previous simulated experiments in the literature [17, 127, 165] to evaluate the simulator. Note that the parameter values used in the following simulation experiments are approximately equal to the parameter set in the existing simulated experiments.

5.1.1 Impact of Number of Pages

The number of pages depicts the number of data items available in the system. Hosting more pages accessible for each transaction leads to less contention for a particular page, which in turn fewer deadlocks occur. The range of the number of pages varied from 40 to 280 pages (that is, 5 to 35 pages per site).

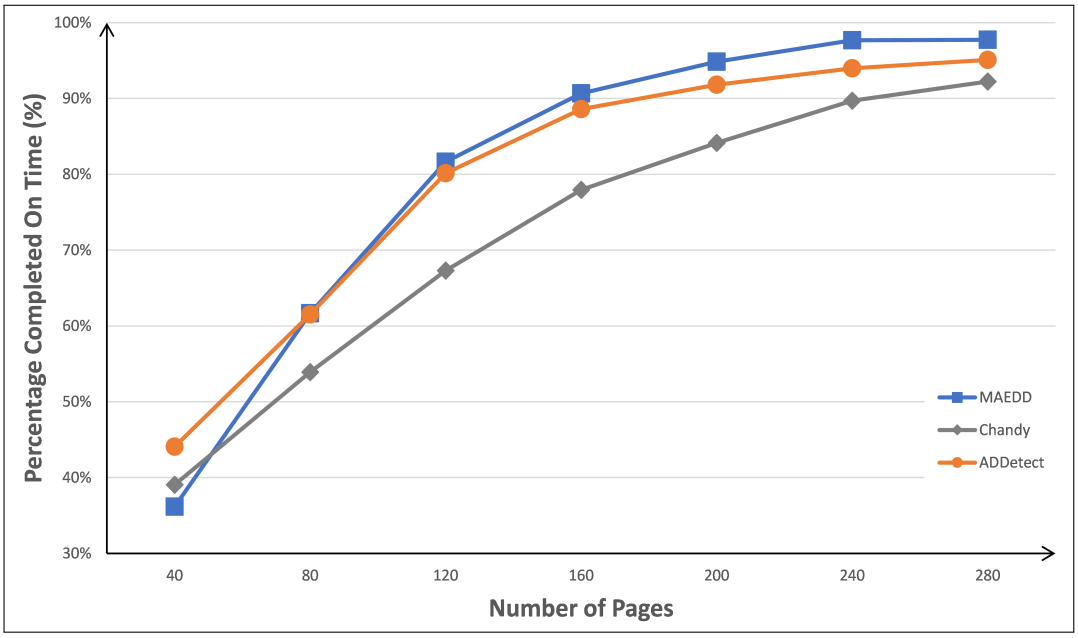


Figure 5.1: PCOT versus Number of Pages

Figure 5.1, shows the comparative PCOT for varying the number of pages available in the system. Note that the transaction update rate is 100% that results in high data contention. The relatively high value of transactions writes operation favor the stationary agent approach, `ADDetect`, when the number of pages is low. Although, in a high number of pages `ADDetect` grows gradually but falls marginally below `MAEDD`. One can note that each agent-based algorithm (`ADDetect` or `MAEDD`) generally outperforms the Chandy algorithm, but note that Chandy achieves a higher PCOT than `MAEDD` when there are a few pages available. However, Chandy

improvement increasingly degrades as the number of pages increases.

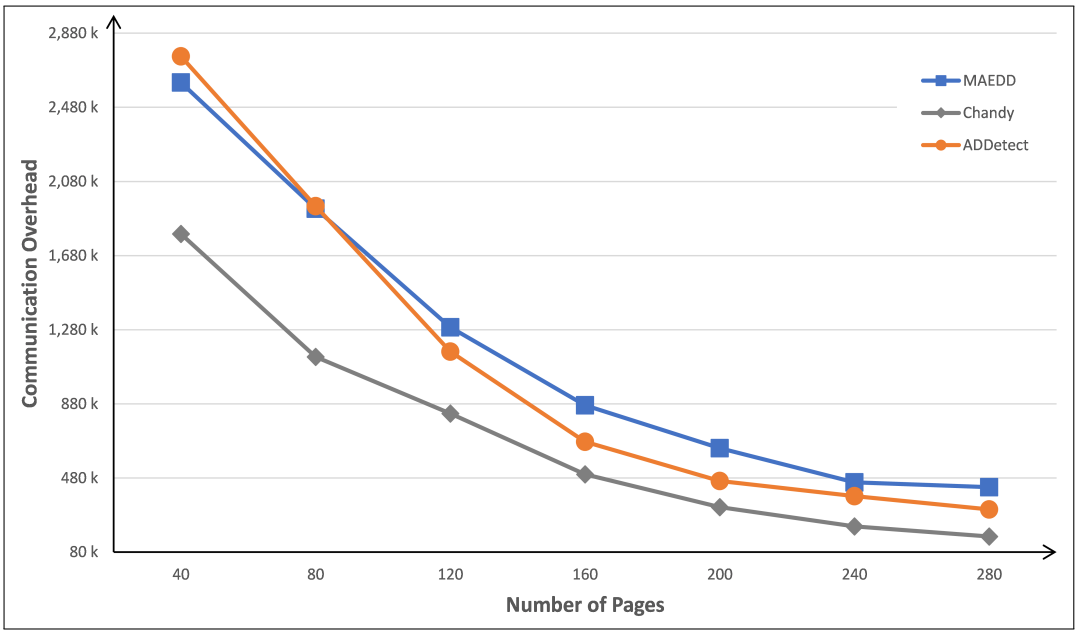


Figure 5.2: Overhead versus Number of Pages

Figure 5.2, illustrates the same set of experiments, but compares communication overhead. The mobile agent approach imposes considerably higher overhead, particularly in situations when the number of pages is high. Generally, `ADDetect` performs more efficiently than `MAEDD`, but fails the mobile agent approach when the number of pages is low. On the other hand, the Chandy algorithm outperforms `MAEDD` and `ADDetect` when it comes to communication overhead.

5.1.2 Impact of Transaction Arrival Interval

The transaction arrival interval denotes the average time difference between the two transactions arriving at a node. When the arrival interval is low, many transactions enter the system within a short duration that increases the chance of data

contention. Longer arrival intervals provide time for more transactions to complete their tasks.

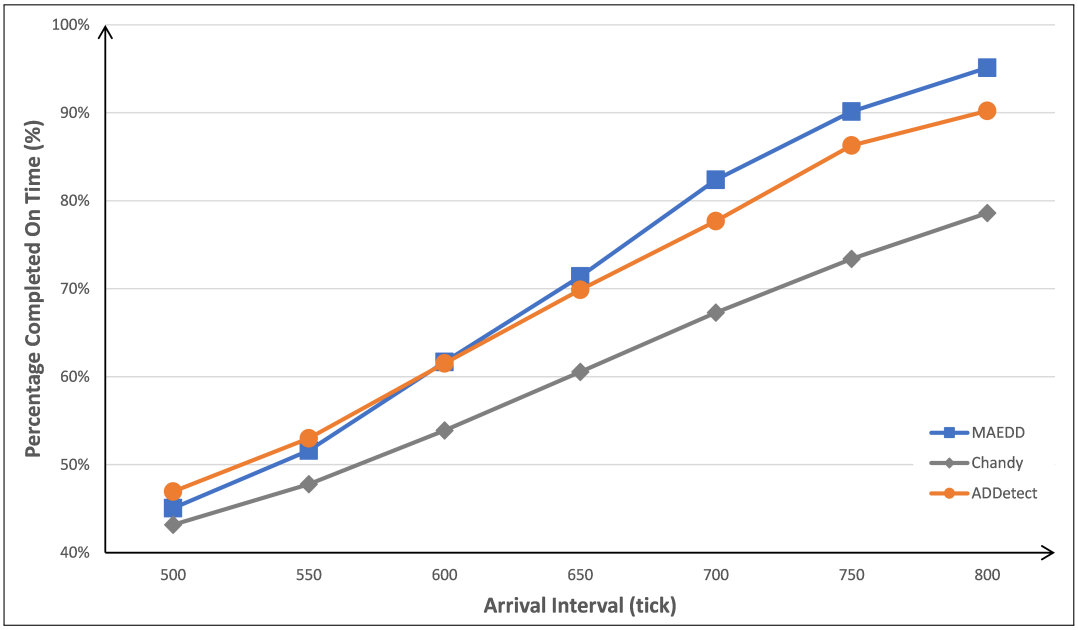


Figure 5.3: PCOT versus Arrival Interval

In this experiment, we observe the comparative PCOT for varying the inter-arrival time of transactions. Figure 5.3 shows the simulation results. As can be seen, the agent-based algorithms, `ADDetect` and `MAEDD`, outperform the edge chasing algorithm, Chandy, in low arrival interval values. The system throughput impact of `MAEDD` becomes manifest only when the mobile agents are more effective, that is, when they have an advantage over stationary agents in particular situations. In this case, when the transactions arrival interval is high, the mobile agent approach is more efficient, hence there is system throughput gain by employing `MAEDD` over `ADDetect` and Chandy. On the other hand, the imposed overhead of employing Chandy outperforms the agent-based algorithms as shown in Figure 5.4.

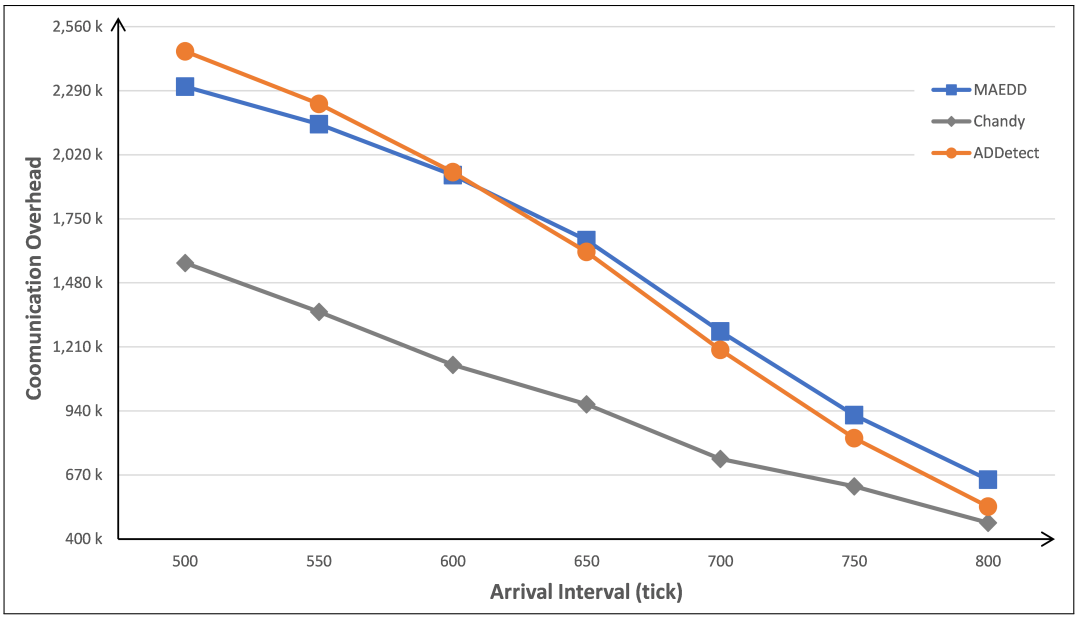


Figure 5.4: Overhead versus Arrival Interval

5.1.3 Impact of Page Update Percentage

The update rate indicates the probability of a page being written to or updated during a transaction's execution. A 0% update rate implies that the transaction executes read operations only, while a 100% rate indicates complete write operations on the transaction's execution. Write operations lock data items exclusively, which block other transactions. Thus, more transaction delay occurs, resulting in low PCOT.

Figure 5.5 shows the PCOT of the system while varying update percentage of transactions. As can be seen, the PCOT is maximum when update percentage is close to 0% because of no blocking transactions. As the update percentage increases, the PCOT drops and eventually decreases gradually to below 65% where all operations in a transaction require a write to the database. In this experiment, the `MAEDD` has an advantage over the `ADDetect` and Chandy one, as low write op-

eration hampers it less significantly while ADDetect is more effective than Chandy when update percentage is high. One can note that MAEDD algorithm degrades significantly as the transaction update rate increases and it achieves almost the same PCOT as ADDetect in 100% write operations.

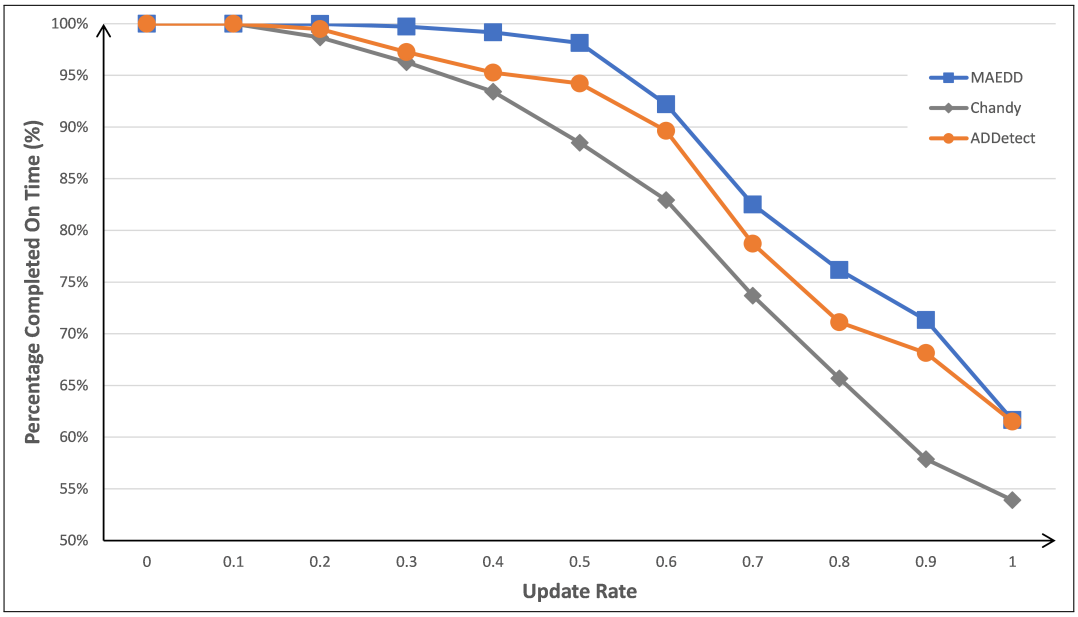


Figure 5.5: PCOT versus Update Percent

Accordingly, it can be seen in Figure 5.6 that there is a significant overhead rise by employing MAEDD compared to ADDetect and Chandy, particularly, when the page writes operations increase.

5.1.4 Impact of Maximum Active Transactions

Maximum active transactions denote the highest number of the transactions running concurrently on a server node. The more operations there are executing simultaneously on a server node, the more data items become blocked, resulting in deadlocks.

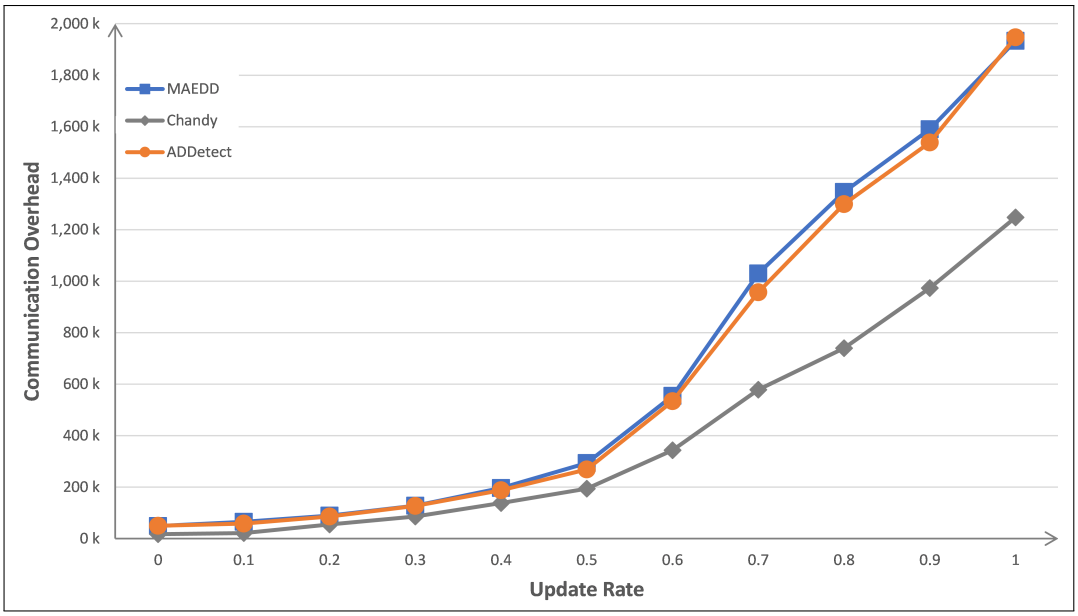


Figure 5.6: Overhead versus Update Percent

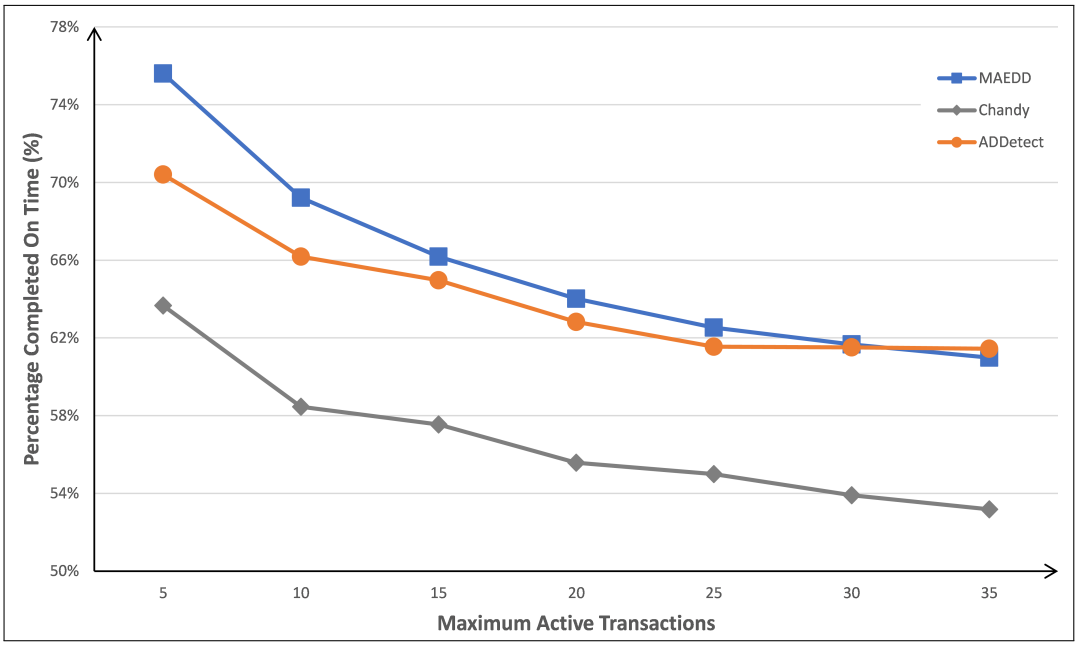


Figure 5.7: PCOT versus Maximum Active Transactions

Figure 5.7 and Figure 5.8 present the results of the experiments, in which we observe the behaviours of Chandy, `MAEDD`, and `ADDetect` by varying the maximum active transactions in the system. We set the highest number of active transactions to a relatively low value of 5 to a relatively high value of 35, as there are no no-

ticeable results outside of this range. As can be seen, the agent-based algorithms outperform the Chandy algorithm. Although, `MAEDD` archives at a higher PCOT when the number of active transactions are low, it underperformed compared to `ADDetect` in situations when the number of active transactions are high. Moreover, it can be seen that the communication overhead increases when the number of transactions increases. One can note that `ADDetect` is more efficient when the number of active transactions is low. However, with the rise of the active transactions, the effect of this advantage decreases, as the stationary agent approach becomes less efficient, *i.e.*, there will be more agents recruited for detecting deadlocks, and a typical agent would interact with the system and other agents.

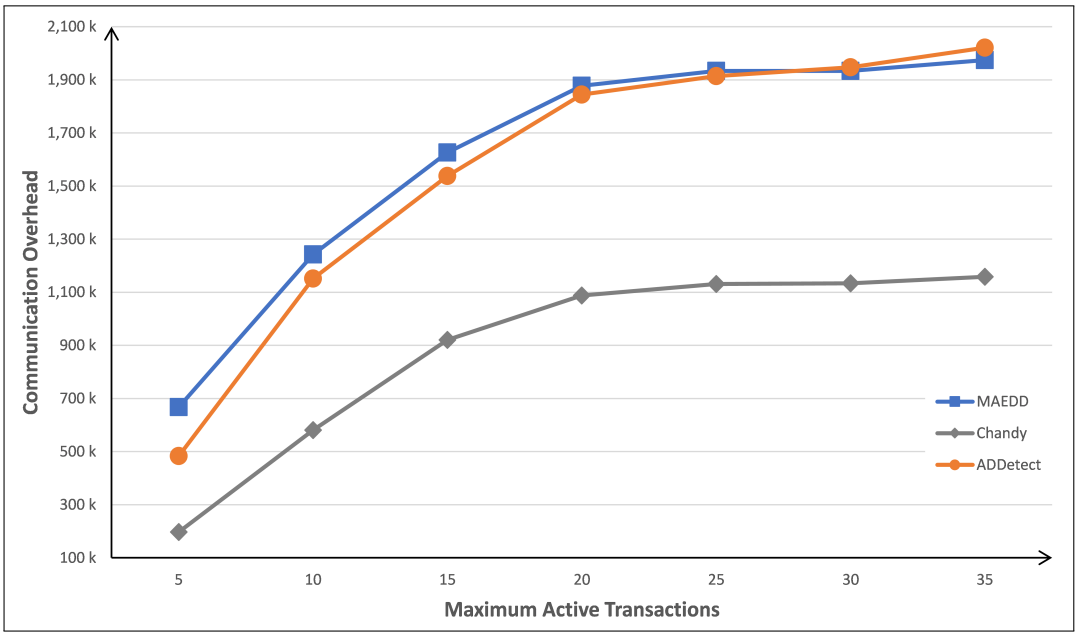


Figure 5.8: Overhead versus Maximum Active Transactions

5.1.5 Impact of Detection Interval

The deadlock detection interval represents the time between each search for deadlock. When the detection interval is small, the deadlock detection algorithm executes more frequently result in identifying deadlocks faster. Although detecting deadlocks faster diminishes the deadlock persistence time, it comes at the cost of extra overhead.

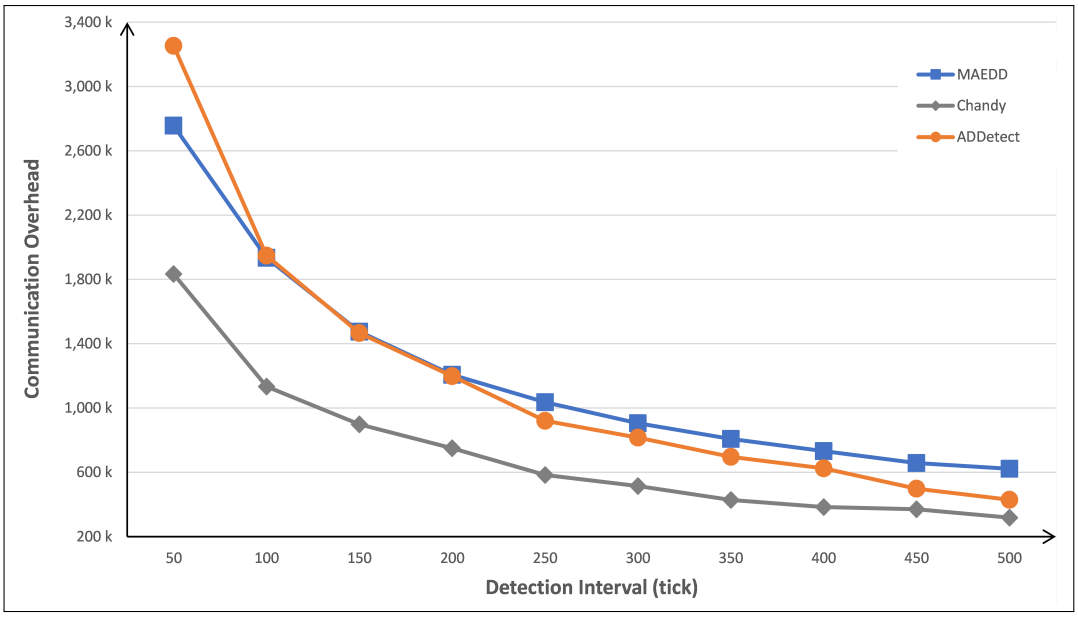


Figure 5.9: Overhead versus Detection Interval

In the last experiment, we observed the comparative performance of Chandy, `MAEDD`, and `ADDetect` while varying the initial deadlock detection interval value. The simulation results are presented in Figure 5.9. As can be seen, the edge chasing algorithm (Chandy) imposes considerably less overhead than agent-based algorithms (`MAEDD` and `ADDetect`). However, the overhead of employing `ADDetect` decreases more in situations when the time interval between detection processes is high and fewer agents can be involved in a detection process, as discussed in Section 3.2.2. For example, it can be observed that `ADDetect` and Chandy dictate

almost the same overhead when there is a high detection interval. Conversely, when the detection interval is low, the MAEDD algorithm has an advantage over ADDetect, thus we observe overhead loss by employing MAEDD over ADDetect. Although a shorter deadlock detection interval increases the PCOT, the results were not significant enough to present.

5.1.6 Summary

The results in this section confirm our previous analysis of the relative advantages and drawbacks of each particular agent-based deadlock detection approach and identify the critical circumstances where one of them dominates significantly. In general, ADDetect outperforms the other algorithms when the workload is high and fails in low workload environments. This analysis of their performance motivates us to experiment Agent Deadlock Resolution Algorithm (ADRes) in a combined design strategy, in which we leverage the strengths of the agent platform in one algorithm.

5.2 The System Performance Impact of ADCombine

In this section, we present experimental results in various scenarios to examine the new combined algorithm, the Agent Deadlock Combined Detection and Resolution Algorithm (ADCombine), which is a combination of ADDetect and ADRes. We compare the performance of Priority Deadlock Resolution (PDR), First Deadlock Resolution (FDR), and ADCombine in identical configurations. In each experiment,

we employ ADDetect as the detection algorithm across the parameter space and the rest of the configuration remains the same as the values selected in Table 5.1.

5.2.1 Impact of Number of Pages

In the first experiment, we vary the number of pages available in the system. Figure 5.10 presents the comparative PCOT achieved by FDR, PDR, and ADCombine. It can be observed that the PDR dominates the agent-based algorithm, ADCombine, when the number of pages is low. This happens because the individual droppability of transactions declines as the number of deadlocks grows, which hampers the proactive offering of opt out (Section 3.2.4). For a moderate to a high number of pages, ADCombine dominates because more agents offer to opt out. Furthermore, ADCombine is more efficient when pages are highly available (Figure 5.11); this describes a situation in which less negotiation is required.

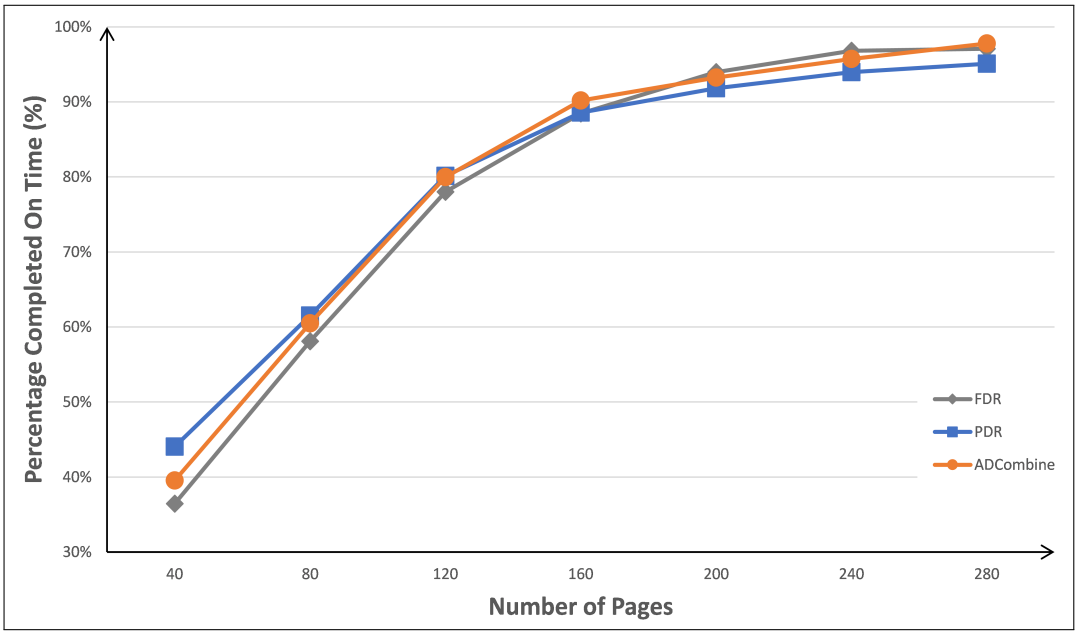


Figure 5.10: PCOT versus Number of Pages

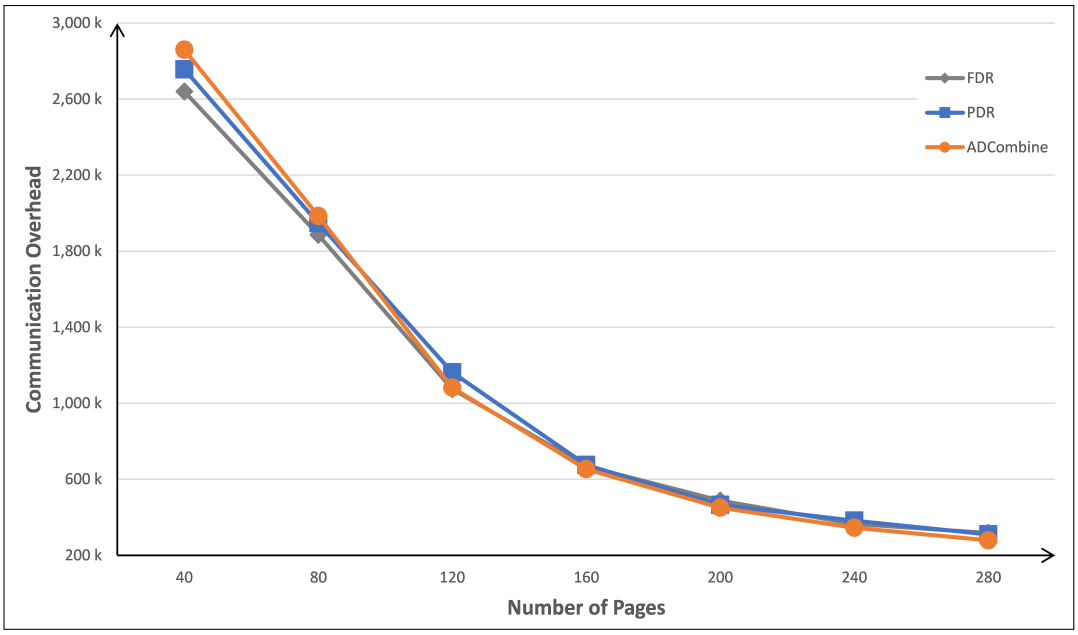


Figure 5.11: Overhead versus Number of Pages

5.2.2 Impact of Transaction Arrival Interval

Figure 5.12 presents the results of the second experiment in which we vary the transaction arrival interval. The comparative PCOT show that the agent-based algorithm, `ADCombine`, dominates when transaction inter-arrival time is high. This is because the agent-based approach is more effective with more time available for negotiation. On the other hand, the PDR algorithm is more efficient at moderate to low transaction arrival times as shown in Figure 5.13. One can note that the FDR algorithm imposes marginally more overhead compared to `ADCombine` and PDR, as the higher priority transactions drop and restart which leads to more deadlocks.

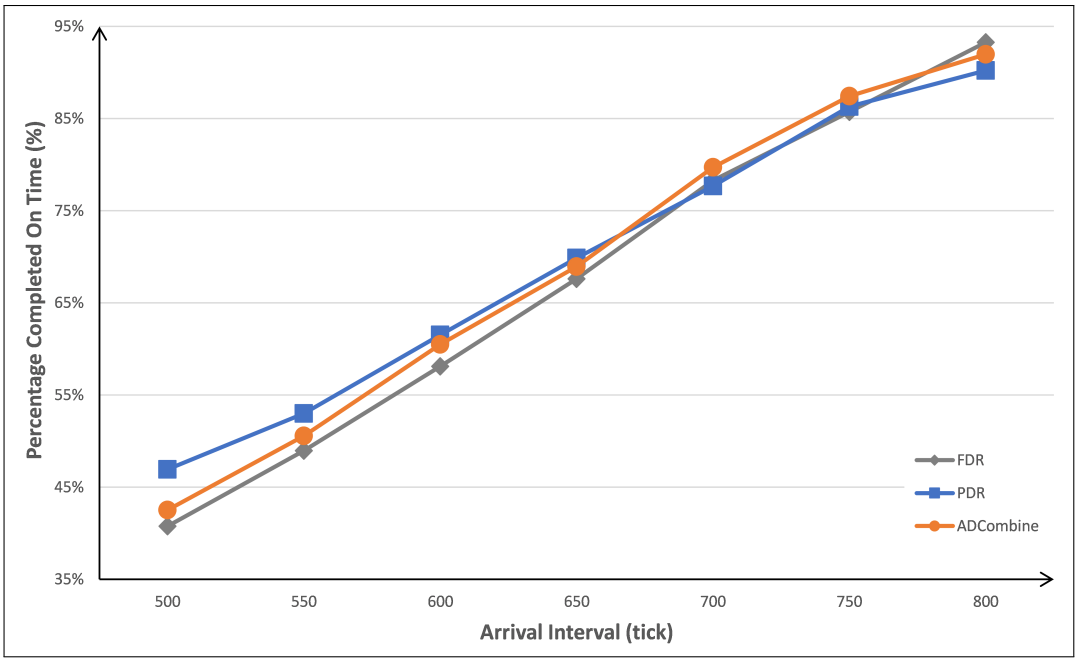


Figure 5.12: PCOT versus Arrival Interval

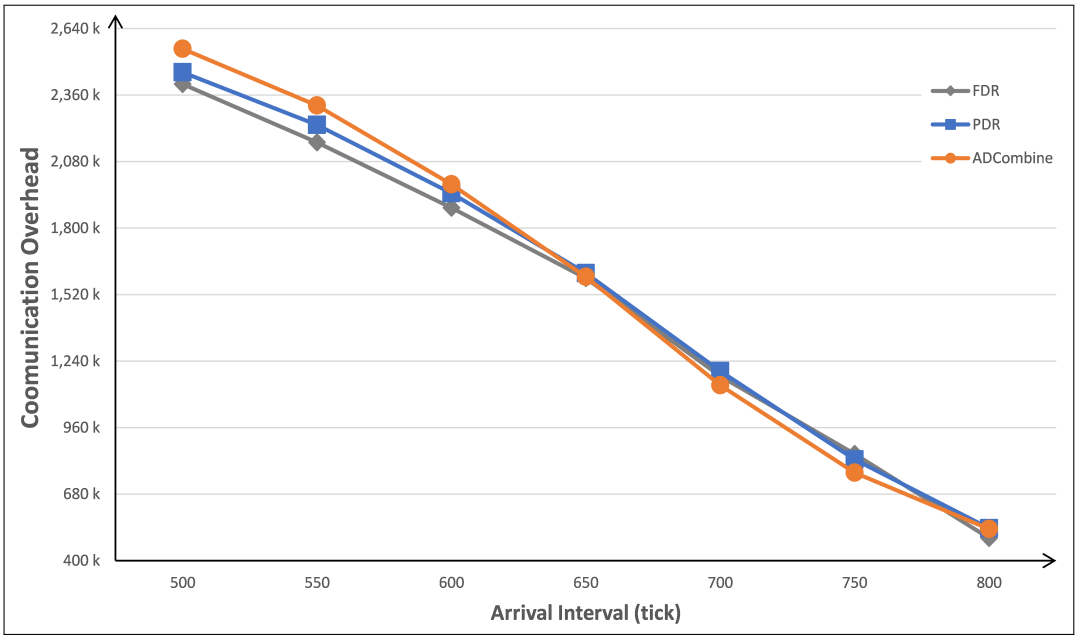


Figure 5.13: Overhead versus Arrival Interval

5.2.3 Impact of Page Update Percentage

In the third experiment, we varied the transaction update percentage. Figure 5.14 presents the results. In the lower update percentage, PCOT is maximum as writing operations are less than reading operation, results in more transactions complete their tasks before the deadline expires. As can be seen, the `ADCombine` dominates when the update percentage is low and achieves almost the same PCOT as PDR in 100% write operations. On the other hand, the efficiency of the FDR algorithm dominates, particularly when the update percentage is high (Figure 5.15).

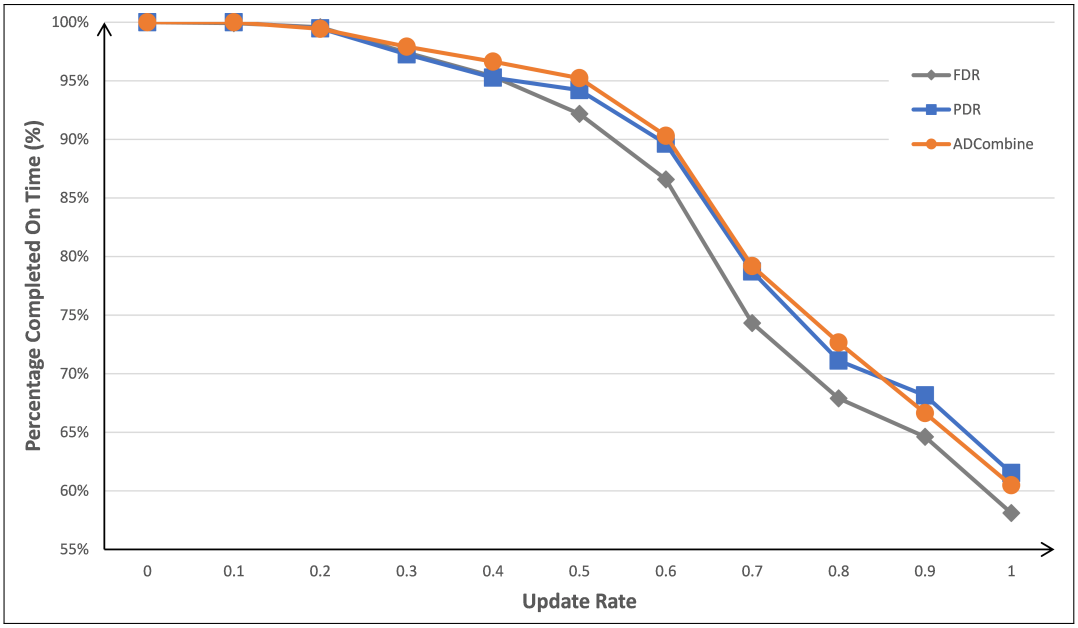


Figure 5.14: PCOT versus Update Percent

5.2.4 Impact of Maximum Active Transactions

Figure 5.16, presents the results of the experiment in which we varied the maximum active transactions on the system. The comparative system throughput

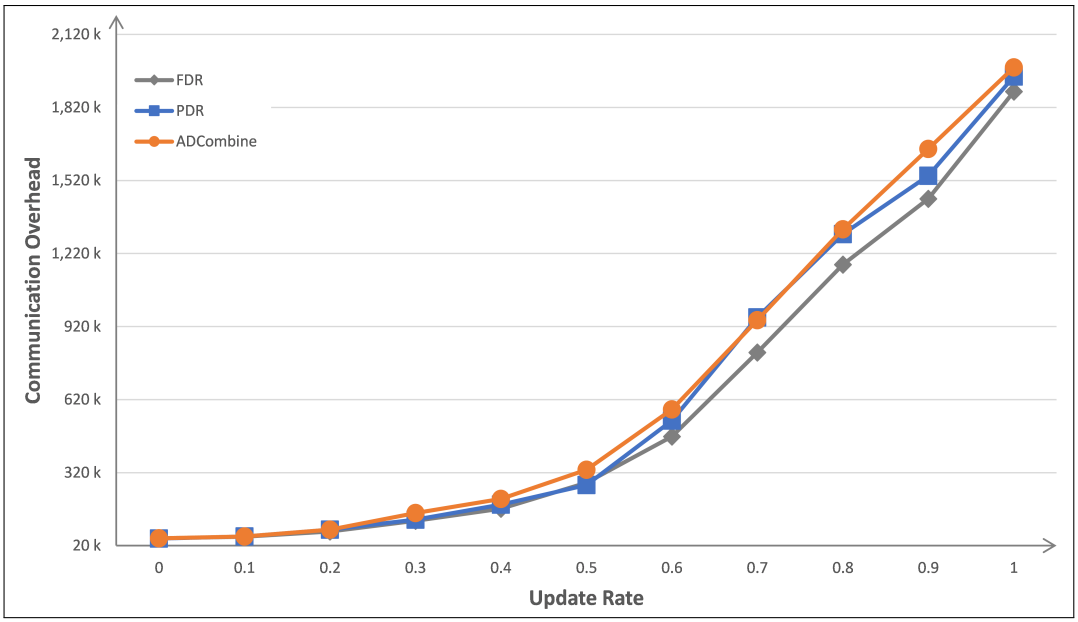


Figure 5.15: Overhead versus Update Percent

shows that `ADCombine` outperforms the two traditional algorithms in most of the parameter space. The exception arises when the number of active transactions is high. This is in agreement with our analysis of the agents with cooperative behaviour in critical situations where the lexicographic order comparison increases the negotiation time (Figure 5.17).

5.2.5 Impact of Detection Interval

In the last experiment, we explore the relative system overhead for varying the deadlock detection interval. Figure 5.18 shows the comparative imposed communication overhead for the three resolution algorithm. The immediate observation is that the algorithm which employs FDR outperforms the other two algorithms in most of the parameter space. This is because the FDR algorithm does not require transaction comparison, thus FDR prevails for being a simple algorithm. In the

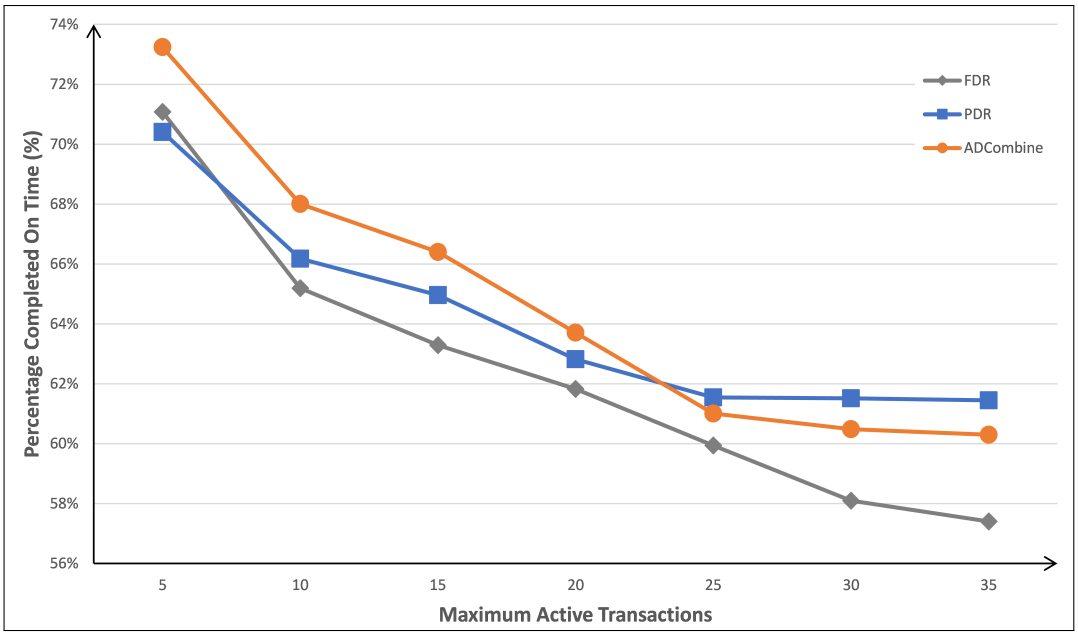


Figure 5.16: PCOT versus Maximum Active Transactions

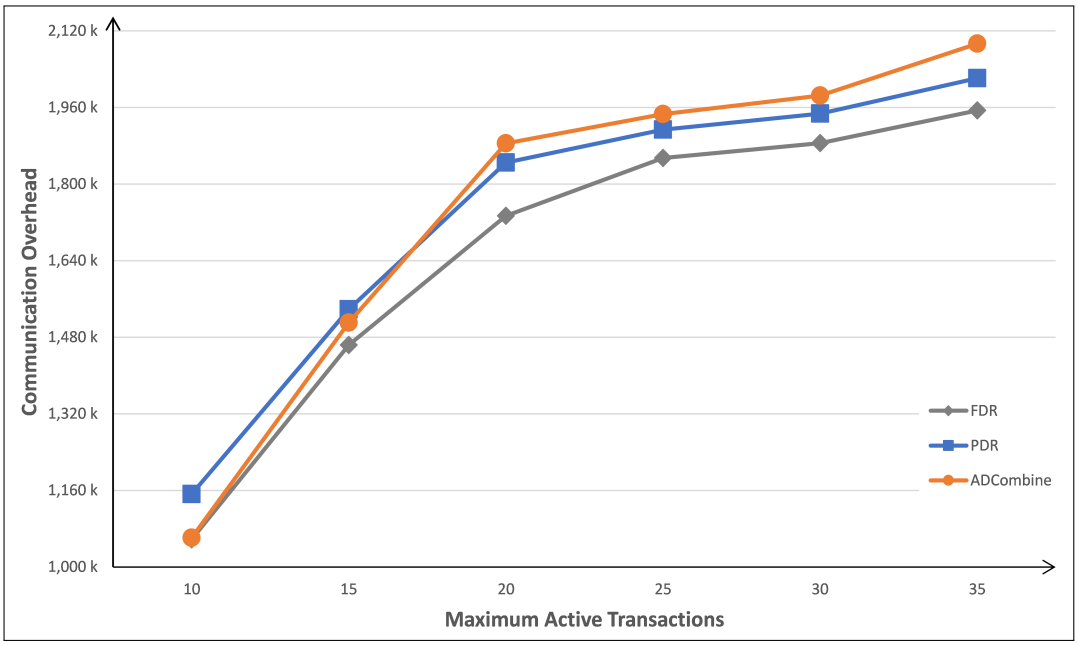


Figure 5.17: Overhead versus Maximum Active Transactions

first value which corresponds to the low detection interval, `ADRes` imposes considerably higher overhead than FDR and PDR. However, it degrades more as the detection interval increases. This suggests the superiority of the combined algorithm, `ADCombine`, which adjusts the interaction and negotiation in agent level.

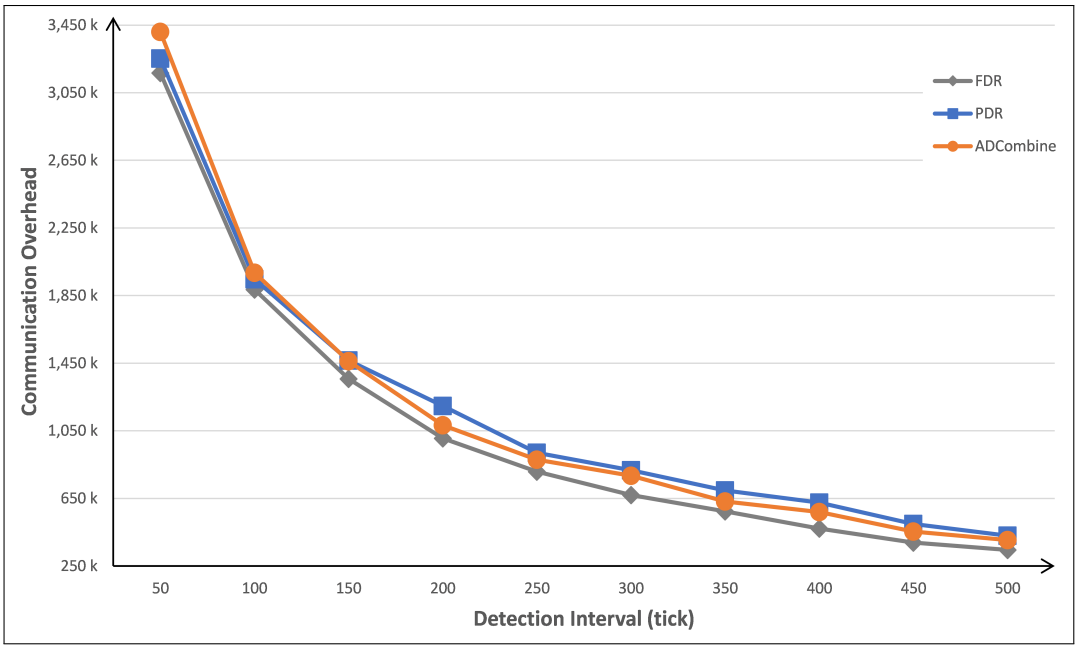


Figure 5.18: Overhead versus Detection Interval

5.2.6 Summary

The results presented in this section confirm that the overall performance of `ADCombine` compared to PDR and FDR is superior in certain situations. It outperforms both traditional algorithms in most of the parameter space, except in the critical situations where there is not enough time for negotiation, and an agent-based approach becomes unproductive. By improving the decision making parameters in Section 3.2.4, we have better tuned the balance between a heavy and a negligible system load, and we better leverage the advantages of the adaptive behaviour in different situations.

Chapter 6

Conclusions and Future Work

Real-Time Database System (RTDBS)s are universally adopted in various practical applications. The increased demand for distributed RTDBS (DRTDBS) in the industry expects an improved system throughput. Deadlock problems can occur in any DRTDBS and has a major impact on the system performance. This thesis presents a new distributed real-time deadlock handling algorithm using agents, called Agent Deadlock Combined Detection and Resolution Algorithm (`ADCombine`), for incorporating helpful behaviour into DRTDBSs. Initial research has comprised a study of literature in several areas of DRTDBS and Multi Agent Systems (MAS), especially deadlock detection and resolution algorithms, helpful behaviour in agent protocols, and deadlock handling using agents. The study led to an observation that, despite the growing use of artificial agents in handling deadlocks for DRTDBS, there is a shortage of algorithms designed for stationary agents, and particularly for incorporating helpful behaviour into stationary agents. Another observation was that some of the existing approaches to deadlock

handling using MAS are either impractical for large DDBSs or over-simplified in their assumptions, which in many realistic environments may be inaccurate. These observations motivated the design of a new algorithm for stationary agents, with particular attention to the cooperation of individual agents involved in the deadlock handling.

Considering the existing algorithms on employing mobile agents in handling deadlocks, Mobile Agent Enabled Deadlock Detection (`MAEDD`), we inherit its principal and design features such as the local execution autonomy. Inspired by previous research on mobile agents, we have analyzed two different approaches to handle deadlocks, including their localization features and decision-making patterns. In the first approach, agents are located in stationary sites and provide services remotely as the need arises; in the second, agents can recruit more agents to decide about which transaction to drop and break the cycle in a bilateral distributed agreement. The novel features of `ADCombine` allow an agent to adjust its execution based on the most up-to-date status of the system. The impact of `ADCombine` on the system's throughput is investigated through an extensive set of simulation experiments using a diverse range of workload and system parameters. The experiment results suggest superiority of `ADCombine` compared to `MAEDD` in certain scenarios.

In our simulation model, we first designed a new detection algorithm using a team of stationary agents with helpful behaviour, called `ADDetect`, to compare with the existing variation of deadlock detection algorithm using MAS. Then, by analysis of the relative strengths and weaknesses of the stationary agents and realizing the possibility that a single algorithm can incorporate an agent-based solution strategy, we have designed the new combined algorithm, `ADCombine`. The new algorithm is a composition of `ADDetect` and `ADRes`, and leverages the advantages

of agents' cooperation and coordination in handling deadlock cycles.

The simulation results demonstrate the system throughput gains for the algorithm that employs stationary agents and mobile agents in high and low data contention respectively. This indicates that higher system loads caused a larger level of congestion, which resulted in the mobile agent thrashing. The `ADDetect` performance is superior in most of the parameter space, except in situations when remote detection and resolution becomes unproductive. This motivated us to introduce an additional strategy into `ADDetect` to employ additional agents to resolve deadlocks more judiciously in a low data contention situation. In the current version of `ADCombine`, the performance was improved in situations where `ADDetect` failed to mobile agent approach.

In this thesis, an agent deliberating about performing deadlock detection relies on its local beliefs, acquired through perception which is assumed to be accurate. Similarly, agents' shared belief is acquired through communication with the rest of the agents. The communication network is assumed to be reliable, and all messages are delivered within a finite amount of time. Also, each agent relies on the transaction execution stability in a detection and resolution process.

6.1 Future Work

We have investigated our agent-based deadlock handling algorithm in the context of a transaction processing, supported in the Distributed Real-Time Database System (DRTDBS). In this study, we have simplified our modeling of the agent

teams, the network, and the transaction execution, to investigate the behaviour of the stationary agent in a relatively stable form, without the impact of extraneous factors that may vary significantly across real-world applications. When applying our algorithms to practical systems, one may need to analyze the engineering requirements such as scalability of changing the size of the system, the possibility of execution fault, communication or execution delay, potential to inject time to the deadlines for completing more transactions, and so forth. Also, the effect of communication overhead in adjusting global detection configuration must considered. Furthermore, implementing agents with organized panic behaviour when the overhead is exponentially increasing can be investigated. Nevertheless, while this thesis did not explore these considerations in particular; this topic leads to interesting research and development questions that merit further study.

Bibliography

[1] Robert K Abbott and Hector Garcia-Molina, *Scheduling real-time transactions: A performance evaluation*, ACM Transactions on Database Systems (TODS) **17** (1992), no. 3, 513–560.

[2] Micah Adler, Eran Halperin, Richard M Karp, and Vijay V Vazirani, *A stochastic process on the hypercube with applications to peer-to-peer networks*, Proceedings of the thirty-fifth annual ACM symposium on Theory of computing, ACM, 2003, pp. 575–584.

[3] Rakesh Agrawal, Michael J Carey, and Miron Livny, *Concurrency control performance modeling: alternatives and implications*, ACM Transactions on Database Systems (TODS) **12** (1987), no. 4, 609–654.

[4] Mumtaz Ahmad, *Query interactions in database systems*, Ph.D. thesis, University of Waterloo, 2013.

[5] Subbu Allamaraju, *Nuts and bolts of transaction processing, white paper*, `http://www.subrahmanyam.com/articles/transactions/NutsAndBoltsOfTP.html`, 1999.

[6] BM Monjurul Alom, Frans Alexander Henskens, and Michael Richard Hannaford, *Deadlock detection views of distributed database*, Information Technology: New Generations, 2009. ITNG'09. Sixth International Conference on, IEEE, 2009, pp. 730–737.

[7] Amos Azaria, Zinovi Rabinovich, Claudia V Goldman, and Sarit Kraus, *Strategic information disclosure to people with multiple alternatives*, ACM Transactions on Intelligent Systems and Technology (TIST) **5** (2015), no. 4, 64.

[8] Jonas Y. Bambi, *Priority-based speculative locking protocols for distributed real-time database systems*, Master's thesis, University Of Northern British Columbia, Prince George, 2008.

[9] Daniel S Bernstein, Robert Givan, Neil Immerman, and Shlomo Zilberstein, *The complexity of decentralized control of markov decision processes*, Mathematics of operations research **27** (2002), no. 4, 819–840.

[10] Philip A Bernstein and Nathan Goodman, *Concurrency control in distributed database systems*, ACM Computing Surveys (CSUR) **13** (1981), no. 2, 185–221.

[11] _____, *An algorithm for concurrency control and recovery in replicated distributed databases*, ACM Transactions on Database Systems (TODS) **9** (1984), no. 4, 596–615.

[12] Philip A Bernstein, Vassos Hadzilacos, and Nathan Goodman, *Concurrency control and recovery in database systems*, Addison- Wesley, 1987.

[13] Philip A Bernstein and Eric Newcomer, *Principles of transaction processing*, Morgan Kaufmann, 2009.

[14] Michael E Bratman, *Intention, plans, and practical reason*, Philosophical Review **100** (1991), no. 2, 277–284.

[15] Michael E Bratman, David J Israel, and Martha E Pollack, *Plans and resource-bounded practical reasoning*, Computational intelligence **4** (1988), no. 3, 349–355.

[16] Jiannong Cao, GH Chan, Tharam S Dillon, et al., *Checkpointing and rollback of wide-area distributed applications using mobile agents*, Parallel and Distributed Processing Symposium., Proceedings 15th International, IEEE, 2001, pp. 1–6.

[17] Jiannong Cao, Jingyang Zhou, Weiwei Zhu, Daoxu Chen, and Jian Lu, *A mobile agent enabled approach for distributed deadlock detection*, International Conference on Grid and Cooperative Computing, Springer, 2004, pp. 535–542.

[18] Sen Cao, Richard A Volz, Thomas R Ioerger, and Michael S Miller, *On proactive helping behaviors in teamwork.*, IC-AI, vol. 5, 2005, pp. 974–980.

[19] K Mani Chandy and Leslie Lamport, *Distributed snapshots: Determining global states of distributed systems*, ACM Transactions on Computer Systems (TOCS) **3** (1985), no. 1, 63–75.

[20] K Mani Chandy and Jayadev Misra, *Asynchronous distributed simulation via a sequence of parallel computations*, Communications of the ACM **24** (1981), no. 4, 198–206.

[21] _____, *A distributed algorithm for detecting resource deadlocks in distributed systems*, Proceedings of the first ACM SIGACT-SIGOPS symposium on Principles of distributed computing, ACM, 1982, pp. 157–164.

[22] K Mani Chandy, Jayadev Misra, and Laura M Haas, *Distributed deadlock detection*, ACM Transactions on Computer Systems (TOCS) **1** (1983), no. 2, 144–156.

[23] Hong-Ren Chen and Yeh-Hao Chin-Fu, *An adaptive scheduler for distributed real-time database systems*, Information Sciences **153** (2003), 55–83.

[24] Shigang Chen and Yibei Ling, *Stochastic analysis of distributed deadlock scheduling*, Proceedings of the twenty-fourth annual ACM symposium on Principles of distributed computing, ACM, 2005, pp. 265–273.

[25] Alok N. Choudhary, Walter H. Kohler, John A. Stankovic, and Don Towsley, *A modified priority based probe algorithm for distributed deadlock detection and resolution*, IEEE Transactions on Software Engineering **15** (1989), no. 1, 10–17.

[26] Philip R Cohen and Hector J Levesque, *Intention is choice with commitment*, Artificial intelligence **42** (1990), no. 2-3, 213–261.

[27] _____, *Teamwork*, Nous **25** (1991), no. 4, 487–512.

[28] Transaction Processing Performance Council, *tpc benchmark b*, Standard Specification, Waterside Associates, Fremont, CA (1990).

[29] _____, *Tpc benchmarks*, 2005.

[30] Saad M Darwish, Adel A El-Zoghabi, and Marwan H Hassan, *Soft computing for database deadlock resolution*, International Journal of Modeling and Optimization **5** (2015), no. 1, 15.

[31] JR González De Mendívil, Federico Fariña, JR Garitagotia, Carlos F Alastruey, and Jose M. Bernabeu-Auban, *A distributed deadlock resolution algorithm for the AND model*, IEEE transactions on parallel and distributed systems **10** (1999), no. 5, 433–447.

[32] Gupta Dhiraj and VK Gupta, *Approaches for deadlock detection and deadlock prevention for distributed systems*, Research Journal of Recent Sciences ISSN **2277** (2012), 2502.

[33] Edsger W Dijkstra and Carel S. Scholten, *Termination detection for diffusing computations*, Information Processing Letters **11** (1980), no. 1, 1–4.

[34] Wanfu Ding and Ruifeng Guo, *Design and evaluation of sectional real-time scheduling algorithms based on system load*, Young Computer Scientists, 2008. ICYCS 2008. The 9th International Conference for, IEEE, 2008, pp. 14–18.

[35] Barbara Dunin-Keplicz and Rineke Verbrugge, *Teamwork in multi-agent systems: A formal approach*, vol. 21, John Wiley & Sons, 2011.

[36] Hywel R Dunn-Davies, RJ Cunningham, and Shamimabi Paurobally, *Propositional statecharts for agent interaction protocols*, Electronic Notes in Theoretical Computer Science **134** (2005), 55–75.

[37] Amr El Abbadi and Sam Toueg, *Availability in partitioned replicated databases*, Proceedings of the fifth ACM SIGACT-SIGMOD symposium on Principles of database systems, ACM, 1985, pp. 240–251.

[38] Gregory D Elder, *Multi-agent coordination and cooperation in a distributed dynamic environment with limited resources: simulated air wars*, Tech. report, DTIC Document, 1993.

[39] Ahmed K Elmagarmid, *Database transaction models for advanced applications*, Morgan Kaufmann Publishers Inc., 1992.

[40] Kapali P. Eswaran, Jim N Gray, Raymond A. Lorie, and Irving L. Traiger, *The notions of consistency and predicate locks in a database system*, Communications of the ACM **19** (1976), no. 11, 624–633.

[41] Amos Fiat, Yishay Mansour, and Uri Nadav, *Efficient contention resolution protocols for selfish agents*, SODA, vol. 7, 2007, pp. 179–188.

[42] Nicholas V Findler and Gregory D Elder, *Multiagent coordination and cooperation in a distributed dynamic environment with limited resources*, Artificial Intelligence in Engineering **9** (1995), no. 3, 229–238.

[43] ACL FIPA, *Foundation of intelligent physical agents*, 2008.

[44] Geoffrey C Fox and Steve W Otto, *Algorithms for concurrent processors*, Physics Today **37** (1984), 50–59.

[45] Stefan Fuenfrocken, *Integrating java-based mobile agents into web servers under security concerns*, System Sciences, 1998., Proceedings of the Thirty-First Hawaii International Conference on, vol. 7, IEEE, 1998, pp. 34–43.

[46] Barbara Gallina, Nicolas Guelfi, and Alexander Romanovsky, *Coordinated atomic actions for dependable distributed systems: the current state in concepts, semantics and verification means*, The 18th IEEE International Symposium on Software Reliability (ISSRE'07), IEEE, 2007, pp. 29–38.

[47] Hector Garcia-Molina, *Using semantic knowledge for transaction processing in a distributed database*, ACM Transactions on Database Systems (TODS) **8** (1983), no. 2, 186–213.

[48] Hector Garcia-Molina and Bruce Lindsay, *Research directions for distributed databases*, ACM SIGMOD Record **19** (1990), no. 4, 98–103.

[49] Michael P Georgeff and Amy L Lansky, *Reactive reasoning and planning.*, AAAI, vol. 87, 1987, pp. 677–682.

[50] Michael Peter Georgeff and Anand S Rao, *A profile of the Australian artificial intelligence institute*, IEEE Expert: Intelligent Systems and Their Applications **11** (1996), no. 6, 89–92.

[51] Christos K Georgiadis and Elias Pimenidis, *Proposing an evaluation framework for B2B web services-based transactions*, Proceedings of the E-Activity and Leading Technologies conference, IASK, Porto, Portugal, 2007, pp. 164–171.

[52] James Gosling, Bill Joy, Guy L Steele, Gilad Bracha, and Alex Buckley, *The java® language specification java se 8 edition*, vol. 8, Pearson Education, 3 2015.

[53] Paul WPJ Grefen and Peter MG Apers, *Dynamic action scheduling in a parallel database system*, International Conference on Parallel Architectures and Languages Europe, Springer, 1992, pp. 807–824.

[54] Barbara J Grosz and Sarit Kraus, *Collaborative plans for complex group action*, Artificial Intelligence **86** (1996), no. 2, 269–357.

[55] Ramesh Gupta, Jayant Haritsa, and Krithi Ramamritham, *Revisiting commit processing in distributed database systems*, ACM SIGMOD Record **26** (1997), no. 2, 486–497.

[56] Theo Haerder and Andreas Reuter, *Principles of transaction-oriented database recovery*, ACM Computing Surveys (CSUR) **15** (1983), no. 4, 287–317.

[57] Jayant R Haritsa, Michael J Carey, and Miron Livny, *Data access scheduling in firm real-time database systems*, Real-Time Systems **4** (1992), no. 3, 203–241.

[58] Jayant R Haritsa and Krithi Ramamritham, *Adding pep to real-time distributed commit processing*, Real-Time Systems Symposium, 2000. Proceedings. The 21st IEEE, IEEE, 2000, pp. 37–46.

[59] Jayant R Haritsa, Krithi Ramamritham, and Ramesh Gupta, *Real-time commit processing*, Real-Time Database Systems, Springer, 2002, pp. 227–243.

[60] Carl S Hartzman, *The delay due to dynamic two-phase locking*, IEEE transactions on software engineering **15** (1989), no. 1, 72–82.

[61] Mark F Hornick and Stanley B Zdonik, *A shared, segmented memory system for an object-oriented database*, ACM Transactions on Information Systems (TOIS) **5** (1987), no. 1, 70–95.

[62] Jiandong Huang, John A Stankovic, Krithi Ramamritham, and Don Towsley, *On using priority inheritance in real-time databases*, Real-Time Systems Symposium, 1991. Proceedings., Twelfth, IEEE, 1991, pp. 210–221.

[63] Sheung-Lun Hung and Kam-Yiu Lam, *Performance comparison of static vs. dynamic two phase locking protocols*, Journal of Database Management (JDM) **3** (1992), no. 2, 12–23.

[64] Dedi Iskandar Inan and Ratna Juita, *Analysis and design complex and large data base using MySQL™ workbench*, International Journal of Computer Science & Information Technology **3** (2011), no. 5, 173.

[65] Sreekaanth S Isloor and T Anthony Marsland, *The deadlock problem: An overview.*, IEEE Computer **13** (1980), no. 9, 58–78.

[66] Takayuki Ito, Hiromitsu Hattori, and Mark Klein, *Multi-issue negotiation protocol for agents: Exploring nonlinear utility spaces*, IJCAI, vol. 7, 2007, pp. 1347–1352.

[67] Hideshi Itoh, *Incentives to help in multi-agent situations*, Econometrica: Journal of the Econometric Society (1991), 611–636.

[68] Sheena Iyengar, *The art of choosing*, Twelve, 2010.

[69] E Douglas Jensen, C Douglas Locke, and Hideyuki Tokuda, *A time-driven scheduling model for real-time operating systems*, RTSS, vol. 85, 1985, pp. 112–122.

[70] Leslie Pack Kaelbling, Michael L Littman, and Anthony R Cassandra, *Planning and acting in partially observable stochastic domains*, Artificial intelligence **101** (1998), no. 1, 99–134.

[71] Ece Kamar, Ya'akov Gal, and Barbara J Grosz, *Incorporating helpful behavior into collaborative planning*, Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems-Volume 2, International Foundation for Autonomous Agents and Multiagent Systems, 2009, pp. 875–882.

[72] Ben Kao and Hector Garcia-Molina, *An overview of real-time database systems*, Real Time Computing, Springer, 1994, pp. 261–282.

[73] Zvi M Kedem and Abraham Silberschatz, *Controlling concurrency using locking protocols*, Foundations of Computer Science, 1979., 20th Annual Symposium on, IEEE, 1979, pp. 274–285.

[74] _____, *Locking protocols: from exclusive to shared locks*, Journal of the ACM (JACM) **30** (1983), no. 4, 787–804.

[75] Edgar Knapp, *Deadlock detection in distributed databases*, ACM Computing Surveys (CSUR) **19** (1987), no. 4, 303–328.

[76] Natalija Krivokapić, *Control mechanisms in distributed object bases*, vol. 54, IOS Press, 1999.

[77] Natalija Krivokapić, Alfons Kemper, and Ehud Gudes, *Deadlock detection in distributed database systems: a new algorithm and a comparative performance analysis*, The International Journal on Very Large Data Bases **8** (1999), no. 2, 79–100.

[78] Shivendra P Kumar, Hari T Krishna, and RK Kapoor, *Distributed deadlock detection technique with the finite automata*, Journal of Information Technology & Software Engineering **5** (2015), no. 2, 1.

[79] Twi-Wei Kuo, Yuan-Ting Kao, and LihChyun Shu, *A two-version approach for real-time concurrency control and recovery*, High-Assurance Systems Engineering Symposium, 1998. Proceedings. Third IEEE International, IEEE, 1998, pp. 279–286.

[80] Kam-yiu Lam, *Concurrency control in distributed real-time database systems*, Ph.D. thesis, City Polytechnic of Hong Kong, 1994.

[81] Kwok-Wa Lam and Sheung-Lun Hung, *A pre-emptive transaction scheduling protocol for controlling priority inversion*, Real-Time Computing Systems and Applications, 1996. Proceedings., Third International Workshop on, IEEE, 1996, pp. 144–151.

[82] Kwok-wa Lam, Sang Hyuk Son, and Sheung-Lun Hung, *A priority ceiling protocol with dynamic adjustment of serialization order*, Data Engineering, 1997. Proceedings. 13th International Conference on, IEEE, 1997, pp. 552–561.

[83] Butler W Lampson and Howard E Sturgis, *Reflections on an operating system design*, Communications of the ACM **19** (1976), no. 5, 251–265.

[84] Jeffrey A Le Pine, Mary A Hanson, Walter C Borman, and Stephan J Motowidlo, *Contextual performance and teamwork: Implications for staffing*, Research in personnel and human resources management **19** (2000), 53.

[85] Victor CS Lee, Kam-Yiu Lam, and Ben Kao, *Priority scheduling of transactions in distributed real-time databases*, Real-Time Systems **16** (1999), no. 1, 31–62.

[86] Thomas C Leonard, *Richard H. Thaler, Cass R. Sunstein, nudge: Improving decisions about health, wealth, and happiness*, Constitutional Political Economy **19** (2008), no. 4, 356–360.

[87] Kristina Lerman and Aram Galstyan, *Agent memory and adaptation in multiagent systems*, Proceedings of the second international joint conference on Autonomous agents and multiagent systems, ACM, 2003, pp. 797–803.

[88] Hector J Levesque, Philip R Cohen, and José HT Nunes, *On acting together*, AAAI, vol. 90, 1990, pp. 94–99.

[89] Tong Li, Carla Schlatter Ellis, Alvin R Lebeck, and Daniel J Sorin, *Pulse: A dynamic deadlock detection mechanism using speculative execution.*, USENIX Annual Technical Conference, General Track, vol. 44, 2005.

[90] Jan Lindström and Tiina Niklander, *Benchmark for real-time database systems for telecommunications*, Databases in Telecommunications II (2001), 88–101.

[91] Yibei Ling, Shigang Chen, and Cho-Yu Jason Chiang, *On optimal deadlock detection scheduling*, IEEE Transactions on Computers **55** (2006), no. 9, 1178–1187.

[92] Chung Laung Liu and James W Layland, *Scheduling algorithms for multiprogramming in a hard-real-time environment*, Journal of the ACM (JACM) **20** (1973), no. 1, 46–61.

[93] Magnus Ljungberg and Andrew Lucas, *The oasis air traffic management system*, Proc. 2nd Pacific Rim Conference on AI (1992), 1–12.

[94] Philip P Macri, *Deadlock detection and resolution in a codasyl based data management system*, Proceedings of the 1976 ACM SIGMOD international conference on Management of data, ACM, 1976, pp. 45–49.

[95] Roger Mailler and Victor Lesser, *Solving distributed constraint optimization problems using cooperative mediation*, Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems-Volume 1, IEEE Computer Society, 2004, pp. 438–445.

[96] Mojtaba Malek Akhlagh, *An interaction protocol for bidirectional deliberation on direct help in agent teamwork*, Master's thesis, University Of Northern British Columbia, Prince George, 2015.

[97] Mojtaba Malek Akhlagh and Jernej Polajnar, *Distributed deliberation on direct help in agent teamwork*, European Conference on Multi-Agent Systems, Springer, 2014, pp. 128–143.

[98] James Mayfield, Yannis Labrou, and Tim Finin, *Evaluation of kqml as an agent communication language*, International Workshop on Agent Theories, Architectures, and Languages, Springer, 1995, pp. 347–360.

[99] David McIlroy and Clinton Heinze, *Air combat tactics implementation in the smart whole air mission model (swarmm)*, Proceedings of the First International SimTecT Conference, 1996.

[100] Maria Miceli, Amedeo Cesta, and Paola Rizzo, *Autonomous help in distributed work environments*, Seventh European Conference on Cognitive Ergonomics, Bonn, Germany, 1994.

[101] Abha Mittal and Sivarama P Dandamudi, *Dynamic versus static locking in real-time parallel database systems*, Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International, IEEE, 2004, p. 32.

[102] C Mohan, Bruce Lindsay, and Ron Obermarck, *Transaction management in the r* distributed database management system*, ACM Transactions on Database Systems (TODS) **11** (1986), no. 4, 378–396.

[103] Aloysius K Mok, *Fundamental design problems of distributed systems for the hard-real-time environment*, Tech. report, Massachusetts Institute of Technology, Cambridge, MA, USA, 1983.

[104] AB MySQL®, *MySQL™: the world's most popular open source database*, `http://www.mysql.com/`, 2005.

[105] Ranjit Nair, Milind Tambe, Makoto Yokoo, David Pynadath, and Stacy Marsella, *Taming decentralized pomdps: Towards efficient policy computation for multiagent settings*, IJCAI, 2003, pp. 705–711.

[106] Ranjit Nair, Pradeep Varakantham, Milind Tambe, and Makoto Yokoo, *Networked distributed pomdps: A synthesis of distributed constraint optimization and pomdps*, AAAI, vol. 5, 2005, pp. 133–139.

[107] Narek Nalbandyan, *A mutual assistance protocol for agent teamwork*, Master's thesis, University Of Northern British Columbia, Prince George, 2012.

[108] Narek Nalbandyan, Jernej Polajnar, Denish Mumbaiwala, Desanka Polajnar, and Omid Alemi, *Requester vs. helper-initiated protocols for mutual assistance in agent teamwork*, 2013 IEEE International Conference on Systems, Man, and Cybernetics, IEEE, 2013, pp. 2741–2746.

[109] Ron Obermarck, *Distributed deadlock detection algorithm*, ACM Transactions on Database Systems (TODS) **7** (1982), no. 2, 187–208.

[110] Shayegan Omidshafiei, Ali-akbar Agha-mohammadi, Christopher Amato, and Jonathan P How, *Decentralized control of partially observable Markov decision processes using belief space macro-actions*, 2015 IEEE International Conference on Robotics and Automation (ICRA), IEEE, 2015, pp. 5962–5969.

[111] Onur Özgün and Yaman Barlas, *Discrete vs. continuous simulation: When does it matter?*, Proceedings of the 27th international conference of the system dynamics society, vol. 6, 2009, pp. 1–22.

[112] M Tamer Özsu and Patrick Valduriez, *Principles of distributed database systems*, Springer Science & Business Media, 2011.

[113] Young Chul Park, Peter Scheuermann, and Sang Ho Lee, *A periodic deadlock detection and resolution algorithm with a new graph model for sequential transaction processing*, Data Engineering, 1992. Proceedings. Eighth International Conference on, IEEE, 1992, pp. 202–209.

[114] Jernej Polajnar, Behrooz Dalvandi, and Desanka Polajnar, *Does empathy between artificial agents improve agent teamwork?*, Cognitive Informatics & Cognitive Computing (ICCI* CC), 2011 10th IEEE International Conference on, IEEE, 2011, pp. 96–102.

[115] Ragunathan Rajkumar, *Synchronization in real-time systems: a priority inheritance approach*, vol. 151, Springer Science & Business Media, 2012.

[116] Anand S Rao, Michael P Georgeff, et al., *Bdi agents: From theory to practice.*, ICMAS, vol. 95, 1995, pp. 312–319.

[117] P Krishna Reddy and Masaru Kitsuregawa, *Speculative locking protocols to improve performance for distributed database systems*, IEEE Transactions on Knowledge and Data Engineering **16** (2004), no. 2, 154–169.

[118] Stuart Jonathan Russell, Peter Norvig, John F Canny, Jitendra M Malik, and Douglas D Edwards, *Artificial intelligence: a modern approach*, vol. 2, Prentice hall Upper Saddle River, 2003.

[119] David Sarne, Avshalom Elmalech, Barbara J Grosz, and Moti Geva, *Less is more: restructuring decisions to improve agent search*, The 10th International Conference on Autonomous Agents and Multiagent Systems-Volume 1, International Foundation for Autonomous Agents and Multiagent Systems, 2011, pp. 431–438.

[120] Srinivasan Selvaraj and Rajaram Ramasamy, *An efficient detection and resolution of generalized deadlocks in distributed systems*, International Journal of Computer Applications **1** (2010), no. 19, 1–7.

[121] Lui Sha, Ragunathan Rajkumar, and John P Lehoczky, *Concurrency control for distributed real-time databases*, ACM SIGMOD Record **17** (1988), no. 1, 82–98.

[122] ———, *Priority inheritance protocols: An approach to real-time synchronization*, IEEE Transactions on computers **39** (1990), no. 9, 1175–1185.

[123] Lui Sha, Ragunathan Rajkumar, Sang Hyuk Son, and C-H Chang, *A real-time locking protocol*, IEEE Transactions on computers **40** (1991), no. 7, 793–800.

[124] Udai Shanker, Manoj Misra, and Anil K Sarje, *Distributed real time database systems: background and literature review*, Distributed and parallel databases **23** (2008), no. 2, 127–149.

[125] Yoav Shoham and Kevin Leyton-Brown, *Multiagent systems: Algorithmic, game-theoretic, and logical foundations*, Cambridge University Press, 2008.

[126] Abraham Silberschatz and ZM Kedam, *A family of locking protocols for database systems that are modeled by directed graphs*, IEEE Transactions on Software Engineering (1982), no. 6, 558–562.

[127] Jayanta Singh and SC Mehrotra, *Performance analysis of a real time distributed database system through simulation*, 15th IASTED International Conf. on Applied Simulation & Modelling, Greece, vol. 1, 2006, pp. 45–50.

[128] Sonia Singh and SS Tyagi, *A review of distributed deadlock detection techniques based on diffusion computation approach*, International Journal of Computer Applications **48** (2012), no. 9, 28–32.

[129] Mukesh Singhal, *Deadlock detection in distributed systems*, Computer **22** (1989), no. 11, 37–48.

[130] Dale Skeen, *Nonblocking commit protocols*, Proceedings of the 1981 ACM SIGMOD international conference on Management of data, ACM, 1981, pp. 133–142.

[131] R Smith, *The contract net protocol: High-level communication and control in a distributed problem solver*, IEEE Trans. on Computers, C **29** (1980), 1104–1113.

[132] Robert D Snelick, *Performance evaluation of hypercube applications: Using a global clock and time dilation*, NIST Interagency/Internal Report (NISTIR)-4630 (1991).

[133] Nadav Sofy and David Sarne, *Effective deadlock resolution with self-interested partially-rational agents*, Annals of Mathematics and Artificial Intelligence **72** (2014), no. 3-4, 225–266.

[134] Alireza Soleimany and Zahra Giahi, *An efficient distributed deadlock detection and prevention algorithm by daemons*, International Journal of Computer Science and Network Security (IJCSNS) **12** (2012), no. 4, 150.

[135] Yumnam Somananda, Y Jayanta Singh, Ashok Gaikwad, and SC Mehrotra, *Management of missed transactions in a distributed system through simulation*, Computer Science and Information Technology (ICCSIT), 2010 3rd IEEE International Conference on, vol. 5, IEEE, 2010, pp. 472–476.

[136] Xiaohui Song and Jane WS Liu, *How well can data temporal consistency be maintained?*, Computer-Aided Control System Design, 1992.(CACSD), 1992 IEEE Symposium on, IEEE, 1992, pp. 275–284.

[137] Selvaraj Srinivasan and Ramasamy Rajaram, *Message-optimal algorithm for detection and resolution of generalized deadlocks in distributed systems*, Informatica: An International Journal of Computing and Informatics **35** (2011), no. 4, 489–498.

[138] John A Stankovic and Wei Zhao, *On real-time transactions*, ACM SIGMOD Record **17** (1988), no. 1, 4–18.

[139] Paul R Stokes, *Design and simulation of an adaptive concurrency control protocol for distributed real-time database systems*, Master's thesis, University Of Northern British Columbia, Prince George, 2007.

[140] Alexander D Stoyenko, *Constructing predictable real time systems*, vol. 146, Springer Science & Business Media, 2012.

[141] Herbert P Sutter, *Independent distributed database system*, September 3 2002, US Patent 6,446,092.

[142] Katia P Sycara, *Argumentation: Planning other agents' plans.*, IJCAI, vol. 89, 1989, pp. 20–25.

[143] Katia P Sycara and Michael Lewis, *Integrating intelligent agents into human teams*, Team Cognition (2004), 203–231.

[144] Robert E Tarjan, *Depth-first search and linear graph algorithms*, SIAM journal on computing **1** (1972), no. 2, 146–160.

[145] Robert E Tarjan and Jan Van Leeuwen, *Worst-case analysis of set union algorithms*, Journal of the ACM (JACM) **31** (1984), no. 2, 245–281.

[146] YC Tay and Rajan Suri, *Choice and performance in locking for databases*, Proceedings of the 10th International Conference on Very Large Data Bases, Morgan Kaufmann Publishers Inc., 1984, pp. 119–128.

[147] Yong Chiang Tay, Rajan Suri, and Nathan Goodman, *A mean value performance model for locking in databases: the no-waiting case*, Journal of the ACM (JACM) **32** (1985), no. 3, 618–651.

[148] Matthew E Taylor, Manish Jain, Christopher Kiekintveld, Jun-young Kwak, Rong Yang, Zhengyu Yin, and Milind Tambe, *Two decades of multiagent teamwork research: past, present, and future*, Collaborative Agents-Research and Development, Springer, 2011, pp. 137–151.

[149] Igor Terekhov and Tracy Camp, *Time efficient deadlock resolution algorithms*, Information Processing Letters **69** (1999), no. 3, 149–154.

[150] Alexander Thomasian and In Kyung Ryu, *Performance analysis of two-phase locking*, IEEE Transactions on Software Engineering **17** (1991), no. 5, 386–402.

[151] Waqar ul Haque, *Transaction processing in real-time database systems*, Ph.D. thesis, Iowa State University, 1993, in Digital Repository.

[152] Özgür Ulusoy, *Research issues in real-time database systems: survey paper*, Information Sciences **87** (1995), no. 1-3, 123–151.

[153] Özgür Ulusoy and Geneva G Belford, *A simulation model for distributed real-time database systems*, Simulation Symposium, 1992. Proceedings., 25th Annual, IEEE, 1992, pp. 232–240.

[154] J Wang, S Huang, and N Chen, *A distributed algorithm for detecting generalized deadlocks*, Tech. report, National Tsing-Hua University, 1990.

[155] Michael Wooldridge, *An introduction to multiagent systems*, John Wiley & Sons, 2009.

[156] Michael Wooldridge and Nicholas R Jennings, *Formalizing the cooperative problem solving process*, Synthese Library (1997), 143–162.

[157] Jie Wu, *Distributed system design*, CRC press, 1998.

[158] Chin-Fu Yeung and Sheung-Lun Hung, *A new deadlock detection algorithms for distributed real-time database systems*, Reliable Distributed Systems, 1995. Proceedings., 14th Symposium on, IEEE, 1995, pp. 146–153.

[159] _____, *Deadlock resolution for distributed real-time database systems*, Microelectronics Reliability **36** (1996), no. 6, 807–820.

[160] Philip S Yu, Kun-Lung Wu, Kwei-Jay Lin, and Sang H Son, *On real-time databases: Concurrency control and scheduling*, Proceedings of the IEEE **82** (1994), no. 1, 140–157.

[161] Liu Yunsheng, *Real time database systems*, Comput Sci **21** (1994), no. 3, 42–46.

[162] Bernard P Zeigler, Herbert Praehofer, and Tag Gon Kim, *Theory of modeling and simulation: integrating discrete event and continuous complex dynamic systems*, second ed., Academic press, Orlando, FL, USA, 2000.

[163] Aidong Zhang and Bharat Bhargava, *Principles and realization strategies of integrating autonomous software systems: extension of multidatabase transaction management techniques*, CSD-TR (1994), 1–19.

[164] Alexander Zharkov, *Performance evaluation of transaction handling policies on real-time dbms prototype.*, SYRCoDIS, 2007.

[165] Jingyang Zhou, Xiaolin Chen, Han Dai, Jiannong Cao, and Daoxu Chen, *M-guard: a new distributed deadlock detection algorithm based on mobile agent technology*, International Symposium on Parallel and Distributed Processing and Applications, Springer, 2004, pp. 75–84.

Appendices

Appendix A

Probabilistic Simulation Model

A.1 Average Expected IO Time of a Distributed Transaction at a Site

The following section provides the details of IO time from the adopted model by Ulusoy *et al.* in [153]. The variables used in the equations are presented in Table A.1.

Table A.1: IO Consumption Variables

Variable	Description
$\bar{t}_{io,local}$ ($\bar{t}_{io,remote}$)	The average IO time expected of a local(remote) transaction T that originated at site s
$\overline{P}$	Average number of data items accessed by each transaction
$\bar{t}_{io,op,local}$ ($\bar{t}_{io,op,remote}$)	Mean IO time expected of the operation submitted by a local(remote) transaction
P_r (P_w)	The probability that T is a read(write) operation
P_L	The probability that the data item p accessed by the transaction has a local copy at site s
$\bar{t}_{read}$ ($\bar{t}_{write}$)	The expected average IO time of executing a read(write) operation
$P_{r,s}$ ($P_{w,s}$)	The probability that remote transaction T accesses the data item p at site s to perform the read(write) operation

A.1.1 Local Transactions

The average IO time expected of a local transaction T that originated at site s specified by $\bar{t}_{io,local}$, can be formulated as:

$$\bar{t}_{io,local} = \overline{P} \times \bar{t}_{io,op,local} \tag{A.1}$$

where $\bar{t}_{io,op,local}$ is defines as:

$$\bar{t}_{io,op,local} = P_r \times P_L \times \bar{t}_{read} + P_w \times P_L \times (\bar{t}_{read} + \bar{t}_{write}) \tag{A.2}$$

A.1.2 Remote Transactions

The average IO time expected of a remote transaction T at site s specified by $\bar{t}_{io,remote}$, can be formulated as:

$$\bar{t}_{io,remote} = \overline{P} \times \bar{t}_{io,op,remote} \tag{A.3}$$

where $\bar{t}_{io,op,remote}$ is defines as:

$$\bar{t}_{io,op,remote} = P_r \times P_{r,s} \times \bar{t}_{read} + P_{r,s} \times P_{w,s} \times (\bar{t}_{read} + \bar{t}_{write}) \tag{A.4}$$

A.2 Average Expected CPU Time of a Distributed Transaction at a Site

The following section provides the details of CPU time from the adopted model by Ulusoy *et al.* in [153]. The variables used in the equations are presented in Table A.2.

Table A.2: CPU Consumption Variables

Variable	Description
$\bar{t}_{cpu,local}$ ($\bar{t}_{cpu,remote}$)	The average CPU time of a local(remote) transaction initialization
$\overline{P}$	Average number of data items accessed by each transaction

Continued on next page

Variable	Description
P_L	The probability that the data item p accessed by the transaction has a local copy at site s
$\bar{t}_{init,local}$ ($\bar{t}_{init,remote}$)	The average CPU time of a local(remote) transaction initialization
$t_{proc,local}$ ($t_{proc,remote}$)	Local(remote) data item process time
$t_{comm,local}$ ($t_{comm,remote}$)	Local(remote) atomic commitment time
t_{locate}	The time required for locating the sites of data items in the access list of a local transaction
$t_{init,cohort}$	The time required for initiating cohorts of a local transaction at remote sits
t_{lookup}	Processing time to locate a single data item
$P_{submit,k}$	The probability that transaction T submits at least one operation to remote site k
$t_{proc,message}$	The CPU time required to process a communication message before being sent of after being received
t_{active}	CPU time required for activating T's operations at remote sites (if remote data access is needed) and processing the corresponding "WORKDONE" messages sent back at the end of each operation execution

Continued on next page

Variable	Description
$\bar{t}_{proc,prep\_to\_comm}$	Average CPU time required for each T's operations due to processing the "PREPARE-TO-COMMIT" message from T's site and sending back "WORKDONE" at the end of each operation execution
$t_{proc,local,p}$ ($t_{proc,remote,p}$)	CPU time required for processing each local(remote) operation
$\overline{N}_{op,submit}$	Average number of operations a local transaction submits to remote sites
$t_{twoPC,phaseOne}$	The CPU time required for the phase 1 of 2PC protocol
$t_{twoPC,phaseTwo}$	The CPU time required for the phase 2 of 2PC protocol
$P_{read,k}$ ($P_{write,k}$)	Probability that T accesses a data item at this site (that is, site k) to perform the read(write) operation

A.2.1 Local Transactions

The average CPU time expected of a local transaction T that originated at site s specified by $\bar{t}_{cpu,local}$, can be formulated as:

$$\bar{t}_{cpu,local} = \bar{t}_{init,local} + t_{proc,local} + t_{comm,local} \tag{A.5}$$

Each of these components is defined as follows:

$$\bar{t}_{init,local} = t_{locate} + t_{init,cohort} \tag{A.6}$$

where t_{locate} is formulated as:

$$t_{locate} = \overline{P} \times t_{lookup} \tag{A.7}$$

and $t_{init,cohort}$'s value for $n-1$ remote data sites is formulated as:

$$t_{init,cohort} = (n-1) \times P_{submit,k} \times t_{proc,message} \tag{A.8}$$

$$t_{proc,local} = t_{active} + \overline{P} \times t_{proc,local,p} \tag{A.9}$$

where t_{active} is formulated as follows:

$$t_{active} = \overline{N}_{op,submit} \times 2 \times t_{proc,message} \tag{A.10}$$

and $t_{proc,local,p}$ is formulated as follows:

$$t_{proc,local,p} = P_L \times \bar{t}_{cpu,local} \tag{A.11}$$

$$t_{comm,local} = (n-1) \times P_{submit,k} \times (t_{twoPC,phaseOne} + t_{twoPC,phaseTwo}) \tag{A.12}$$

where $t_{twoPC,phaseOne}$ is the CPU time required for the phase 1 of 2PC protocol.

Recall that the *prepare phase* of 2PC protocol is sending a message to each of the cohort sites and processing the messages sent back from those sites (Section 2.1.6). It is formulated as follows:

$$t_{twoPC,phaseOne} = 2 \times t_{proc,message} \tag{A.13}$$

and $t_{twoPC,phaseTwo}$ is the CPU time required for the phase 2 of 2PC protocol. Remember that the *commit phase* of 2PC protocol is sending the final decision message to cohort sites (Section 2.1.6). It is formulated as follows:

$$t_{twoPC,phaseTwo} = t_{proc,message} \tag{A.14}$$

A.2.2 Remote Transactions

The average CPU time expected of a remote transaction T at site s specified by $\overline{CPU}_r$, can be formulated as:

$$\bar{t}_{cpu,remote} = \bar{t}_{init,remote} + t_{proc,remote} + t_{comm,remote} \tag{A.15}$$

Each of these components is defined as follows:

$$\bar{t}_{init,remote} = P_{submit,k} \times t_{proc,message} \tag{A.16}$$

$$t_{proc,remote} = \overline{P} \times (\bar{t}_{proc,prep\_to\_comm} + t_{proc,remote,p}) \tag{A.17}$$

where $\bar{t}_{proc,prep\_to\_comm}$ is formulated as follows:

$$\bar{t}_{proc,prep\_to\_comm} = (P_r \times P_{read,k} + P_w \times P_{write,k}) \times (2 \times t_{proc,message}) \tag{A.18}$$

and $t_{proc,remote,p}$ is formulated as follows:

$$t_{proc,remote,p} = (P_r \times P_{read,k} + P_w \times P_{write,k}) \times \bar{t}_{cpu,remote} \tag{A.19}$$

$$t_{comm,remote} = P_{submit,k} \times (t_{twoPC,phaseOne} + t_{twoPC,phaseTwo}) \tag{A.20}$$

where $t_{twoPC,phaseOne}$ and $t_{twoPC,phaseTwo}$ are formulated as follows:

$$t_{twoPC,phaseOne} = 2 \times t_{proc,message} \tag{A.21}$$

$$t_{twoPC,phaseTwo} = t_{proc,message} \tag{A.22}$$

A.3 Agent Deadlock Handling Model

Theorem 1. *An agent will trace a directed graph if the ID of each edge in the cycle is greater than the ID of head edge. Head edge is the initial edge in a tracing process. Hence,*

each deadlock cycle is detected by one and only one agent.

Proof. A deadlock cycle in a distributed database system can be categorized into local and global. A local deadlock is when all the blocked transactions in the cycle reside in one site. On the other hand, a global deadlock consists of multiple sites involved in one cycle. Assume that Figure A.1 presents a deadlock cycle in a database system.

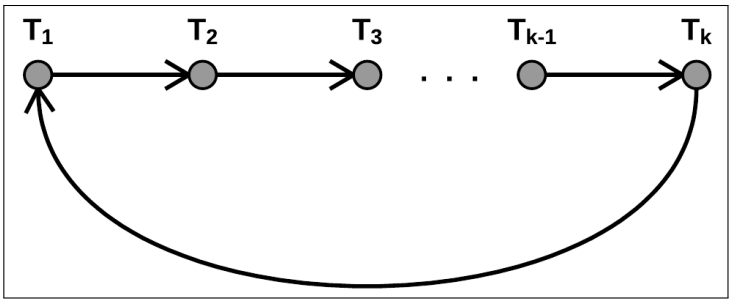


Figure A.1: A worst case example for Tarjan's algorithm [145]

Let $\mathcal{T} = \{T_1, T_2, \cdots, T_k\}$ denote the unique set of k transactions involved in the cycle. Note that each transaction in the database system is assigned for a unique identifier called transaction ID. As transaction ID is a natural number, it exists a transaction in $\mathcal{T}$ such that its ID is lower than any other transaction's ID in the set. So:

$$\exists!\ T_i \in \mathcal{T}, \qquad where \quad ID(T_i) < \bigcup_{j=1, j\neq i}^{k} ID(T_j)\ :\ T_j \in \mathcal{T}$$

We know that in a global deadlock detection process, the transactions are uniquely distributed among active global agents. Let $Active\_GAg = \{GAg_i \mid GAg_i \in Global\ Agents\}$ denote the list of active global agents and $AgT_i = \{T_i \mid T_i \in \mathcal{T}\}$ indi-

cate the list of transactions that GAg_i is responsible for. Thus:

$$\exists!\ GAg_i \in Active\_GAg, \qquad where \quad T_i \in AgT_i$$

Since T_i has the lowest ID in the cycle, then GAg_i is the only global agent that can find the cycle.

In a local deadlock situation, the cycle is handled by one agent only. Because there is only one transaction in a deadlock that holds the lowest ID in compare to other transactions in the cycle, the deadlock can be found only once. □

Theorem 2. *In a deadlock detection process, if* k *is the number of agents involved to detect deadlocks in a graph with* n *nodes, the total computation cost in the worst case scenario would be:*

$$\textit{number of messages} = T(e) \times n \times \frac{k+1}{2}$$

where $T(e)$ *indicates messages transmission time unit.*

Proof. In `ADCombine`, each agent will trace $[\frac{n}{k}]$ transactions in each detection process in the worst case scenario. So, the cost of tracing for the first agent would be n where n denotes the edges of the directed wait-for-graph. The second agent will trace part of the biggest deadlock excluding the part that first agent is responsible. So, the cost of tracing for the second agent would be $n - [\frac{n}{k}]$. The same idea applies to the next agents. Thus, k's agent will trace part of the biggest deadlock excluding the parts that $k-1$ agents are responsible (*i.e.*, $n-(k-1)[\frac{n}{k}]$). Therefore, the number

of traces required to find all the deadlocks in a global deadlock would be:

$$\begin{aligned}
\text{number of messages} &= \sum_{i=0}^{k-1} \left(n - i[\frac{n}{k}]\right) \\
&= n \sum_{i=0}^{k-1} \left(1 - i[\frac{1}{k}]\right) \\
&= n\left(k - \sum_{i=1}^{k-1} \left([\frac{i}{k}]\right)\right) \\
&= n\left(k - \frac{\frac{k(k-1)}{2}}{k}\right) \\
&= n\left(k - \frac{k-1}{2}\right) \\
&= n(\frac{k+1}{2})
\end{aligned} \tag{A.23}$$

Considering $T(e)$ as the time unit required for transmitting one message, then:

$$\therefore \text{ number of messages} = T(e) \times n \times \frac{k+1}{2}$$

where $k > 1$. □

A.3.1 Agent Memory and Adaptation

In this study, we experiment an extension of modeling approach that allows us to incorporate history or memory of agents' perceptions into the mathematical model of a Multi Agent System. An agent's memory is in the form of First-In-First-Out (FIFO) queue data structure with the finite size.

Each agent stores the number of deadlocks occurred in each round along with the detection time. The x-axis is the time unit (tick). The results are used to adjust

the agents' behaviour to the system's current configuration. In this case, we see that as the simulation runs, the deadlock detection/resolution condition evolve to a steady state in which the number of deadlocks at each interval does not change, even though individual agents continue to process. In general, the adaptive agents appear to be more efficient than the non-adaptive agents [87].

We quantify the system's efficiency more precisely by deadlock ratio trend. The deadlock ratio trend is the system alignment an agent will predict for the decisions in the future. The trend is used to adjust the detection interval, number of active agents, and influence the deadlock resolver agents. The deadlock ratio function defines as follows:

$$\delta = \sum_{i=1}^{n} \frac{t_i d_i - n\bar{t}\bar{d}}{t_i^2 - \bar{t}^2} \tag{A.24}$$

where δ is the deadlock ratio, t is the detection time, n is the memory depth, and d is the number of deadlocks detected. $\tan\delta$ is used to identify the trend of the system. The value of this metric is expected to be high when there are many deadlocks, and conversely, it is low, when there is a significant number of unnecessary agents.

A.4 Simulation UML Diagrams

A sample package specification of DRTDBS simulator is presented in Figure A.2. The simulator deadlock handling architecture diagram illustrated in Figure A.3.

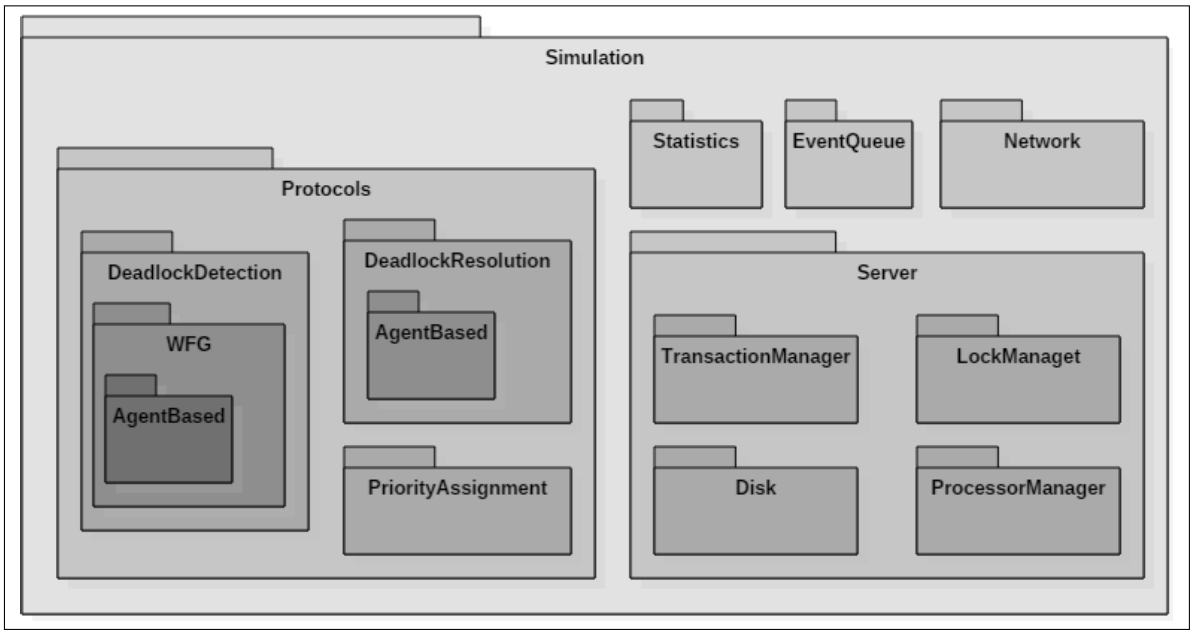


Figure A.2: Simulation Package Diagram

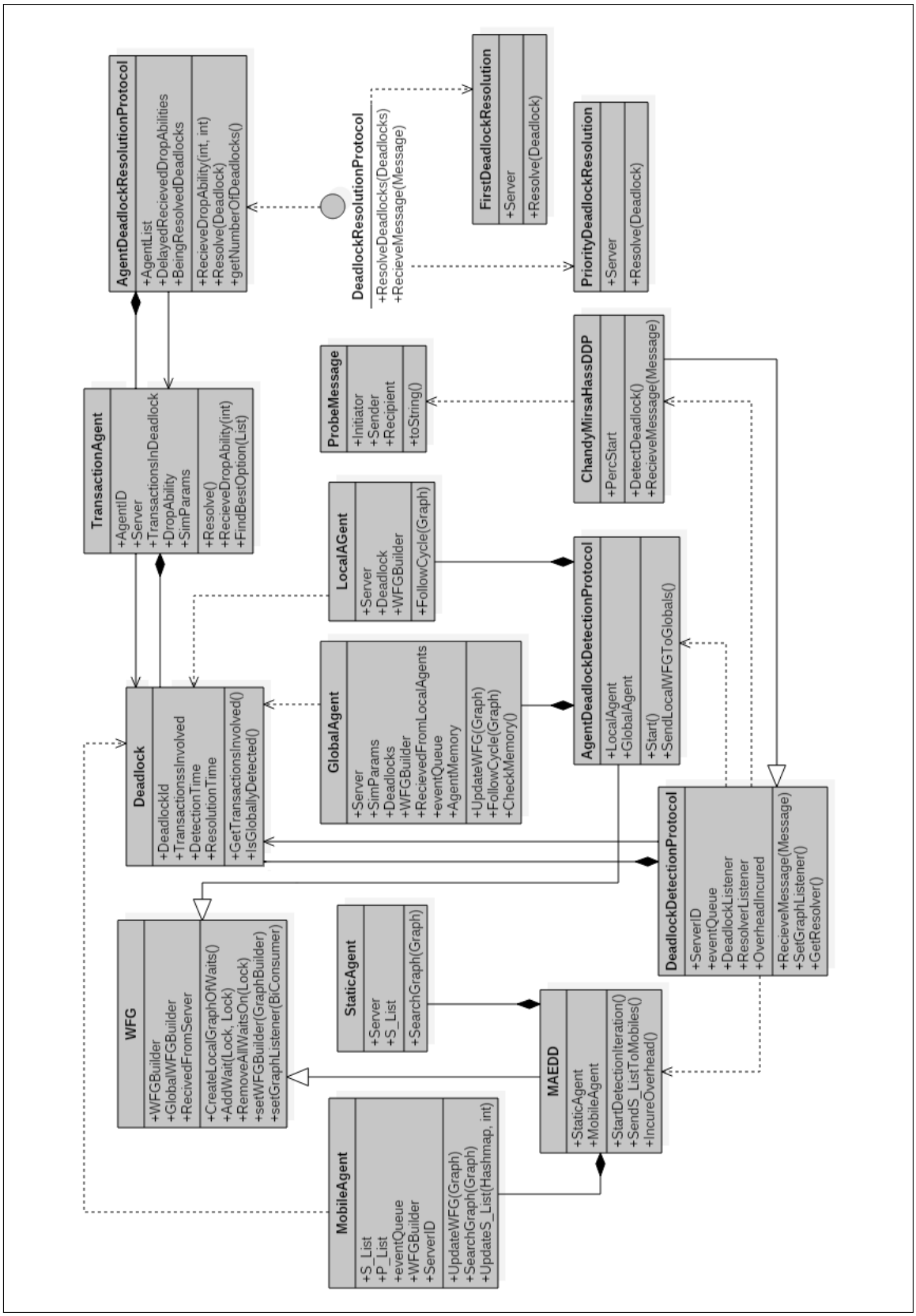


Figure A.3: Simulation Deadlock Handling UML Class Diagram

Appendix B

Deadlock Handling Cost

This section presents the analysis of distributed deadlock scheduling from [24] that used to adjust deadlock detection interval.

B.1 Mathematical Formulation

Theorem 3. *Suppose deadlock formation follows a Poisson process with rate* λ*. The long-run mean average cost of deadlock handling, denoted by* $C(T)$*, is*

$$C(T) = \frac{C_D}{T} + \frac{\lambda \int_0^T C_R(t)dt}{T} \tag{B.1}$$

where the frequency of deadlock detection interval is $1/T$*, and* C_D *and* $C_R(t)$ *denote a deadlock detection and resolution cost respectively.*

Proof. The proof of Theorem 3 is available in [24]. □

Theorem 3 results in the following lemma which demonstrates that in a deadlock detection and resolution process there is an optimal deadlock detection interval, $1/T^*$, that minimizes the deadlock handling overhead in a long-run.

Lemma 4. *Suppose that the message complexity of deadlock detection is $\mathcal{O}(n^\alpha)$, and that of deadlock resolution is $\mathcal{O}(n^\beta)$. If $\alpha < \beta$, there exists a unique deadlock detection frequency ($1/T^*$) that yields the minimum long-run mean average cost when n is sufficiently large. [24]*

Proof. Differentiating Equation (B.1) yields

$$C'(T) = -\frac{C_D}{T^2} + \frac{\lambda C_R(T)}{T} - \frac{\lambda \int_0^T C_R(t)dt}{T^2} \tag{B.2}$$

Define a function $\varphi(T)$ as follows:

$$\varphi(T) \equiv T^2 C'(T) = -C_D + \lambda C_R(T) - \lambda \int_0^T C_R(t)dt \tag{B.3}$$

Notice that $C'(T)$ and $\varphi(T)$ share the same sign. Differentiating $\varphi(T)$, we have

$$\varphi'(T) = \lambda T C'_R(T) \tag{B.4}$$

Because $C_R(T)$ is a monotonically increasing function, $C'_R(T) > 0$, which means $\varphi'(T) > 0$. Therefore, $\varphi'(T)$ is also a monotonically increasing function. $C_R(T) -$

$C_R(t) \leqslant 0$ holds iff $T \leqslant t$. For any given $0 < \xi < T$, it has

$$\begin{aligned} TC_R(T) - \int_0^T C_R(t)dt &= \int_0^T \big(C_R(T) - C_R(t)\big)dt \\ &> \int_0^\xi \big(C_R(T) - C_R(t)\big)dt > \int_0^\xi \big(C_R(T) - C_R(\xi)\big)dt \\ &= \xi\big(C_R(T) - C_R(\xi)\big) \end{aligned} \tag{B.5}$$

Applying Equation (B.5) to Equation (B.3), we have

$$\begin{aligned} \varphi(T) &= -C_D + \lambda\big(C_R(T) - \int_0^T C_R(t)dt\big) \\ &> -C_D + \lambda\xi\big(C_R(T) - C_R(\xi)\big) \end{aligned} \tag{B.6}$$

We further have

$$\begin{aligned} \varphi(T) &> -C_D + \lambda\xi C_R(T)\big(1 - \frac{C_R(\xi)}{C_R(T)}\big) \\ &= -C_D + \lambda\xi C_R(T)\theta \end{aligned} \tag{B.7}$$

where $\theta = \big(1 - {}^{C_R(\xi)}/_{C_R(T)}\big)$ and $0 < \theta < 1$ since $C_R(T)$ is monotonically increasing. Substituting $C_D = c_1 n^\alpha$ and $C_R(\infty) = c_2 n^\beta$ in Equation (B.7), we obtain

$$\lim_{T\to\infty} \varphi(T) > -c_1 n^\alpha + \lambda\xi\theta c_2 n^\beta \tag{B.8}$$

Since $\alpha < \beta$, $\lim_{T\to\infty} \varphi(T)$ is asymptotically dominated by the term $\lambda\xi\theta c_2 n^\beta$ when n is sufficiently large. Observe that $\varphi(0) = -C_D < 0$, and $\varphi(T)$ is monotonically increasing. By the intermediate value theorem, it must be true that there

exists a unique T^*, $0 < T^* < \infty$, such that

$$\varphi(T) = T^2 C'(T) = \begin{cases} < 0, & 0 \leqslant T < T^* \\ = 0, & T = T^* \\ > 0, & T > T^* \end{cases} \tag{B.9}$$

It means that $C(T)$ reaches its minimum at and only at $T = T^*$. The existence and the uniqueness of optimal deadlock detection interval $T^* = \arg\big(\min_{T>0} C(T)\big)$ is proved. [24] □