DISTRIBUTED SLIDING WINDOW SCHEDULING IMPLEMENTED IN JAVA FOR A JADE AGENT SYSTEM

by

Scott Logie

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF APPLIED SCIENCE

In the School
of
Engineering Science

SIMON FRASER UNIVERSITY

Spring 2005

APPROVAL

NAME:	Scott Logie
DEGREE:	Master of Applied Science
TITLE OF THESIS:	Distributed Sliding Window Scheduling Implemented in Java for a JADE Agent System

EXAMINING COMMITTEE

Chair:

Dr. William A. Gruver
Academic Supervisor
Professor, School of Engineering Science

Dorian Sabaz
Technical Supervisor
Intelligent Robotics Corporation

Dilip Kotak
Committee Member
National Research Council

Dr. John Dill
Internal Examiner
Professor, School of Engineering Science

Date Approved: November 24, 2004

ABSTRACT

This thesis describes a fully distributed approach to resource scheduling within a sliding time frame, implemented for a system of agents across multiple JADE platforms. All agents, with operations inside the current window, schedule tasks using recursive propagation and a sorting algorithm. Operations outside the window are not scheduled until either the sliding window has advanced to encompass them or until gaps have opened between tasks inside the sliding window to accommodate them. The distributed sliding window approach to scheduling addresses many of the problems afflicting both centralized systems, including scalability, robustness, and responsiveness to dynamic changes. It also provides full decentralization as compared with other distributed approaches.

ACKNOWLEDGEMENTS

This research has been made possible by the generous financial support of the Natural Sciences and Engineering Research Council of Canada, the BC Advanced Systems Institute, the Instrumentation, Systems, and Automation Society, and Simon Fraser University.

I would like to thank Dr. William A. Gruver and Dorian Sabaz for their valuable input and guidance throughout my research and development.

Most importantly, thanks to Joy for her unwavering support.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

GLOSSARY

ACL	Agent Communication Language
AID	Agent Identifier
API	Application Program Interface
CSE	Composite Scheduling Entity
FIPA	Foundation for Intelligent Physical Agents
HTTP	HyperText Transfer Protocol
IDEA	Intelligent and Distributed Enterprise Automation
JADE	Java Agent Development Framework
JXTA	Sun Microsystems framework for peer-to-peer communication
LAN	Local Area Network
SDK	Java Standard Development Kit
SFU	Simon Fraser University
VNET	Virtual Network Project of the IDEA Laboratory

1 INTRODUCTION

Traditional techniques for scheduling resources suffer from three major disadvantages: limited responsiveness to dynamic changes, large expense related to scalability, and poor robustness. Many of the standard approaches do not respond well to a changing work environment, become increasingly complex for systems of large dimension, and use client/server architectures that may fail when components become unavailable. Instead, distributed approaches to scheduling have emerged as a potential solution to these problems. This thesis presents one such method.

The document will begin with an overview of scheduling and existing centralized approaches. It will then describe the benefits that distributed system designs provide and outline a new distributed algorithm for scheduling. In the results provided here and the analysis that follows, this algorithm will be shown to greatly reduce communication and computational requirements.

1.1 Basic Scheduling

According to Baker [1], scheduling is "the allocation of resources over time to perform a collection of tasks". Schedulers assign tasks to available resources, and then determine a sequence and chronological placement of those tasks that satisfy the constraints. As a result, a generic scheduling model can be used to represent many industries and applications, from manufacturing processes to the power distribution, from human interactions in a workplace to the allocation of transport vehicles in a shipping fleet.

Regardless of the actual items being scheduled, common nomenclature has been established to describe these problems. In scheduling problems, *jobs* (also called *orders* or *parts*), each consisting of a series of *operations* (or *tasks*) that have an associated precedence sequence, are assigned to a number of available *resource* entities and given chronological placement. For simple scheduling problems, few system details need to be specified apart from the *machine function* and finite *processing time* associated with each task, which introduce constraints on the problem of resource allocation.

The machine function of a resource refers to a specific action that the device performs to complete its task. Though many resources may be available to handle tasks, only some may be equipped to perform the specific machine function associated with each. As a result, the machine function parameter limits the number of available resources to which the task may be assigned. Likewise, this parameter limits the number of resources to which a task may be reassigned in the event that one of the other shop resources goes off-line. Additionally, systems may be comprised of both single-function and multi-function resources. Many algorithms have been developed to handle the allocation of tasks to resources given the presence of both single-function and multi-function machines [2][3]. In the example of Figure 1, the available task can be assigned to either of resources Alpha or Beta because both are equipped to handle drilling.

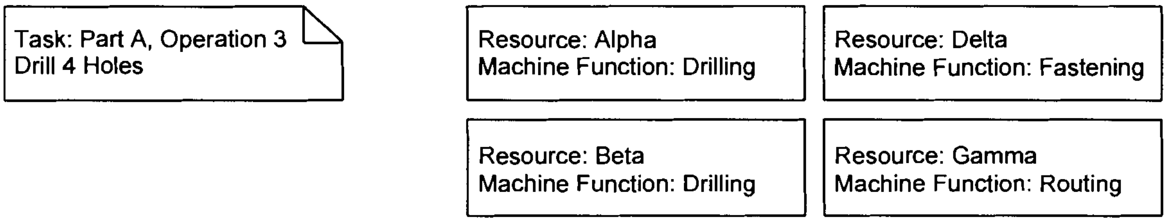

Task: Part A, Operation 3
Drill 4 Holes
Resource: Alpha
Machine Function: Drilling
Resource: Delta
Machine Function: Fastening
Resource: Beta
Machine Function: Drilling
Resource: Gamma
Machine Function: Routing

FIGURE 1. ILLUSTRATING THE SIGNIFICANCE OF MACHINE FUNCTIONS

Though task allocation is an important component to scheduling, the details of how tasks are assigned to resources are not the focus of the work presented here. Instead, this thesis focuses on task sequencing and task timing.

1.2 Standard Approaches to Scheduling

Inherently, the scheduling of resources is a distributed problem. Resources, capabilities, and information about jobs, system components, and states are dispersed among the many, often physically separate, elements of the system. Even so, most present-day scheduling systems, whether manufacturing-based or otherwise, are designed using some form of client/server architecture (Figure 2) with centralized algorithms that construct a global model of the production domain before computing a scheduling solution. Information regarding resources and tasks is collected by a central component, one of a variety of algorithms applied, and the resulting solution redistributed to the rest of the system. Three key requirements are demanded of a central server: large data storage capabilities, significant processing capabilities, and reliable communication paths between the server and peripheral system components.

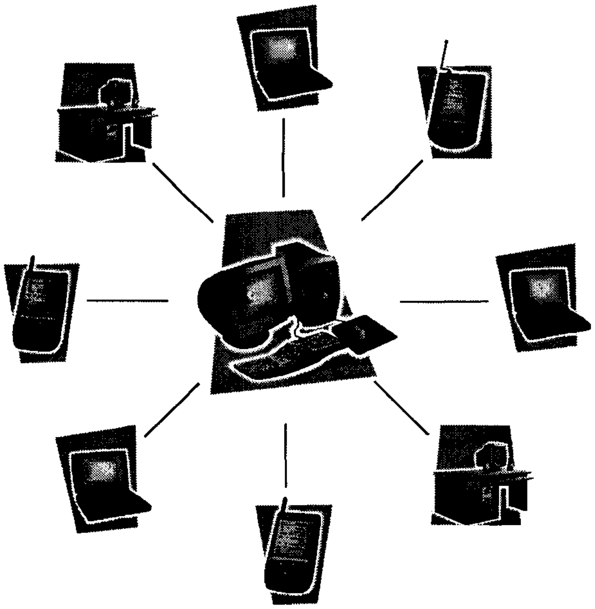


FIGURE 2. CLIENT/SERVER SYSTEM ARCHITECTURE

Many centralized approaches to scheduling have been studied that provide near optimal solutions to the scheduling problem.

Linear programming approaches [4][5] introduce slack variables for each inequality constraint and iteratively increase variables from zero as long as the effect on the objective function is positive. The result is an exploration of the extreme points of the surface bounded by linear constraints until an optimal objective value is found. As problems become increasingly complex, linear programming becomes a less useful approach and further constraint reasoning methods may be applied. These techniques require search methods to find feasible solutions (usually non-optimal) to realistically modeled constrained systems [6].

A further branch-and-bound method explores the enumeration tree of all possible scheduling solutions, but reduces the scope of this exploration by eliminating the nodes for which all descendant solutions must exceed some bound [7][8]. These methods incur a high computational cost as the number of constraints and variables increases. Additionally, all system constraint information, the number of variables, and the relative weight of each variable on the objective function must be predetermined before the algorithm can be applied.

Other research has employed genetic algorithms [9][10], which begin with an initial population of possible solutions and then crossover, mutate, or replace these solutions in an effort to preserve the best aspects of all possible solutions. The more iterations the algorithm completes, the better these solutions are meant to converge on the optimum. Simulated annealing algorithms are another approach [11][12], which always accept solution changes that lead to improved results, but only accept detrimental changes according to some probability function that decreases over time. Like many heuristic methods, the computational requirements are reduced, but it can be difficult to measure the proximity of the optimized solution to the optimum that exists for the problem. Also, if the characteristics of the parent population change, the algorithm must be re-applied.

Further still, Lagrangian relaxation techniques can also be used to obtain near optimal solutions to scheduling problems. System constraints are relaxed using Lagrange multipliers, and the resulting problem is solved using dynamic programming or generalized gradient search methods. Iteratively, the multipliers are adjusted to find a solution that minimizes the objective function while still satisfying all constraints [13].

To optimize schedules with respect to the number of resources, force-directed techniques have emerged to minimize the number of resources required to perform all tasks [14]. For each operation and control step, a force value is calculated. The operations are then placed according to least force to ensure that all tasks are uniformly distributed.

1.2.1 Weaknesses of Standard Approaches

A major failing of centralized algorithms is their dependence on the reliable performance of a single component and the consistent operation of the communication links between server and clients. Should the server crash for any reason or lose contact with its peripheral components, the entire system would be crippled. Centralized communication methods also have inherent reliability issues. If any single communication link between server and client breaks, the affected system component becomes unreachable.

Apart from robustness issues, centralized systems also suffer from an increase in algorithm complexity that accompanies any significant increase in system variables, often making them impractical for large systems. Moreover, since centralized algorithms often use “snapshot” logic, requiring all system variables (including the number of available resources or the number of orders) to be known before a solution can be computed, any change to these variables requires the data to be recollected and the algorithm to be re-applied. As stated by John Layden [15]:

> The trouble is, for discrete manufacturing, what's optimal one minute is obsolete the next when the snapshot changes – for example, the next order arrives or the next change occurs in material supply or resource availability.

These conditions make centralized problem solving approaches neither dynamic nor scalable and costly to implement [16]. As a result, for large and ever-changing systems, centralized approaches become impractical, since many real-world applications of scheduling algorithms do not provide the static environments that these algorithms require to be effective. Instead, variables are constantly changing: new orders arrive and new parts are designed. Meanwhile resources fail, increase in numbers, or are upgraded to include new functions. These environments demand solutions that are scalable to handle changing numbers of orders and resources, yet dynamic enough to allow the details of these elements to change.

Heuristic algorithms begin to address these issues, in that updated variables can be incorporated into subsequent iterations of the algorithm. But a centralized heuristic algorithm still suffers the same reliability limitations of other centralized algorithms and the communication requirements are even greater, since clients are queried with each iteration.

1.3 Distributed Systems

Distributed systems begin to address these issues [2][3][16][17][18][19]. In a vast majority of industries, resources and data are distributed. Separate computers and servers store and process data and the individuals who use these machines possess data and "processing power" of their own. On production plant floors, different machines have different capabilities, functions, and settings. In business dealings, customers know their needs, suppliers know their capabilities, and communication channels exist between parties to find common ground.

In contrast to the centralized approaches described above, distributed systems are those comprised of *agents*, autonomous entities that possess the ability to plan and execute their own actions (Figure 3). In some distributed systems, agents are motivated by self-serving goals. In others, agents collaborate to fulfill group objectives. The goals of the system have a significant impact on the design of agents and their interactions [16]. Regardless, distributed systems rely on agents that base decisions on a limited view of their environment and, through message-passing and negotiation, utilize the other agents at their disposal to obtain the additional information they require.

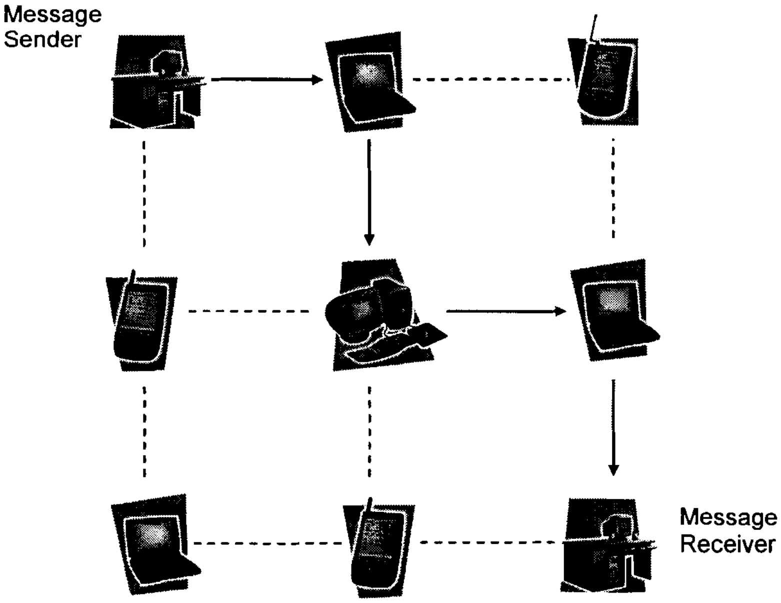


FIGURE 3. DISTRIBUTED SYSTEMS MODEL

Agent-based systems address issues of robustness, since no single component becomes critical to system operation. If a single component fails or becomes unreachable, the other agents are still fully capable of coordinating with their remaining peers by relaying messages through their neighbours, as in Figure 4.

In these systems, the emphasis is on distributing the processing and data storage requirements more equitably among all agents at the expense of the communication required. Although communication becomes increasingly important in distributed systems, distributed communication architectures, such as peer-to-peer networking, make no path critical to system operation. In the event of a broken communication link, messages can be re-routed to reach their destinations. The result is a system that is more scalable, robust, and responsive to dynamic changes than one based on centralization.

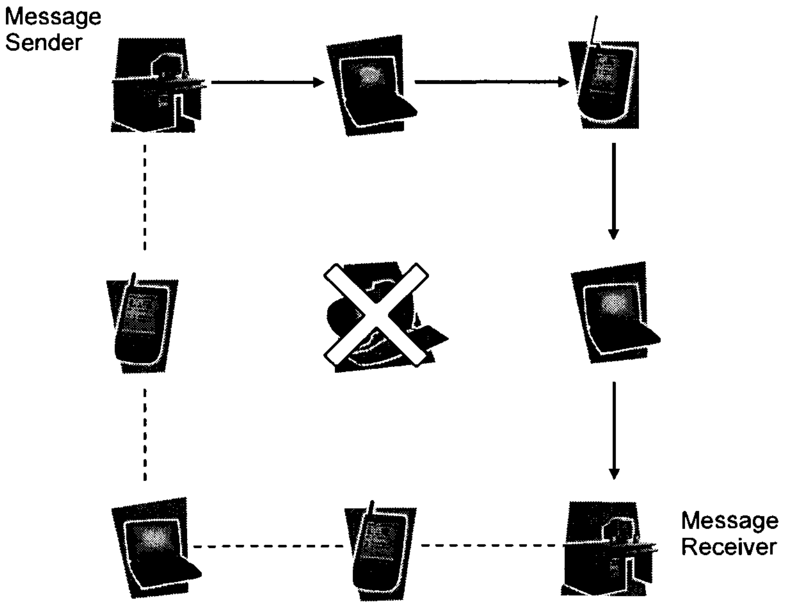


FIGURE 4. CRITICAL COMPONENTS IN DISTRIBUTED SYSTEMS

As the popularity and usage of wireless technologies grows, the types of systems in which component knowledge and resources are dispersed continues to increase. As a result, distributed approaches to system design have become more relevant. But despite the fact that resources in production, manufacturing, transportation, power, and other industries are distributed, not all methods used to organize and manage them are distributed.

For the purpose of this thesis, the term *distributed* will refer not only to the logistical arrangement of resources and data, but also to the technique used to coordinate resources and data. A completely *distributed* system would be distributed in all aspects of its design: the storage of data and information, the communication mechanisms utilized, and the processing performed to execute the algorithm.

In the area of distributed scheduling, multi-agent solutions have been proposed [1][3][17] that assign agents to each resource and, in some cases, to each order on the system. These agents negotiate the schedule among themselves in an effort to optimize performance criteria, often *utilization* of resources (the percentage of time that machines remain busy) or *makespan* (the total production time of the system).

While many of these systems successfully decentralize the processing and data storage of their respective algorithms, not all agent systems provide full distribution of communication. Systems such as those proposed in [1], [3], and [17] use a shared *blackboard* to handle inter-agent communication (Figure 5). The blackboard component serves as a common location for partial solutions to be shared and for agents to exchange information. The deficiency of these systems is that the blackboard agent itself becomes a centralized component in an otherwise distributed system and constitutes a single critical element to system operation. So, while providing great improvements through parallel processing and distributed data storage, these systems are still hindered by one shortcoming of centralized methods: reliance on a singular critical component.

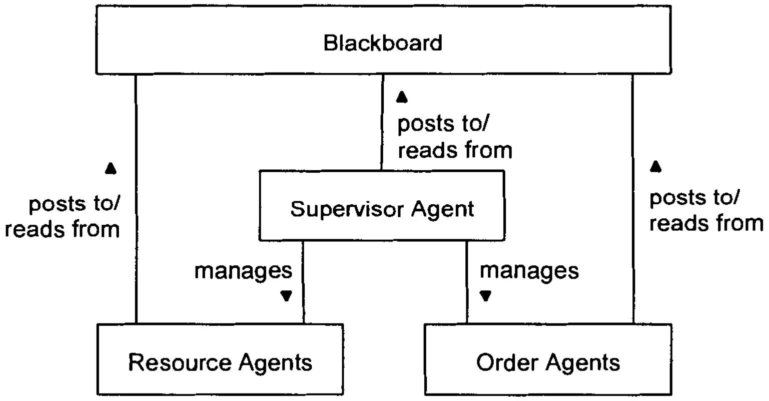


FIGURE 5. BLACKBOARD MULTI-AGENT SYSTEMS

In contrast, Hino, et al., developed the recursive propagation technique for scheduling [18][19], which provides little in terms of decision-making functionality (apart from conflict resolution), but fully distributes the scheduling communication between agents. Agents inform one another of changes to their schedules, propagate these changes to other resources, and return the resulting impact to the initiator of the change (Figure 6). Because Hino's technique can be applied over a peer-to-peer network, it provides the potential for complete decentralization.

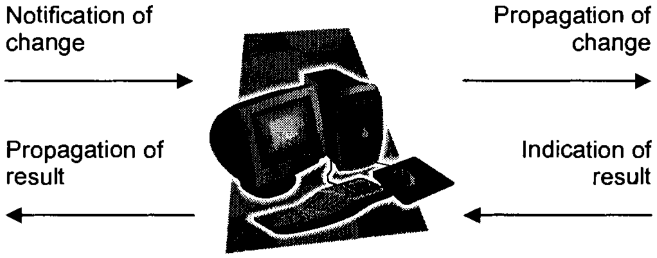


FIGURE 6. RECURSIVE PROPAGATION

1.3.1 Tools for Distributed System Implementation

Significant research has been done in the area of multi-agent system design, and many industrial players have begun to realize the potential in distributed approaches to their endeavours. As a result, the Foundation for Intelligent Physical Agents (FIPA, www.fipa.org) was established in 1996 to provide standards under which development in this area can progress. Universities and industrial partners throughout the world collaborate through FIPA to develop and maintain these standards in agent architectures, communications, management, message transport, and applications,

Based on FIPA compliance, many development tools have been created to aid in the design and implementation of multi-agent systems. Like many of these tools, the Java Agent Development Framework (JADE), developed by Telecom Italia [20], provides both a model for agent architectures and a structure for inter-agent communications. It simplifies the design of multi-agent systems by providing the basic infrastructure on which all multi-agent systems depend. Because it is based on the Java programming language, it also provides improved interoperability between multiple platforms, operating systems, and devices.

Evaluation by Vrba [21] showed that JADE provides faster message throughput, a more stable environment, and superior memory usage to some of its rival systems, like FIPA-OS. Because of these features, JADE was selected as the framework for initial implementation.

1.4 Thesis Objective

To address the problems of scalability, robustness, and responsiveness to dynamic changes that accompany the standard approaches to scheduling, a completely distributed approach to scheduling was desired; one that would not only distribute the processing requirements of the algorithm, but also the communication infrastructure as well. While the recursive propagation technique would address many of these concerns, further agent intelligence would have to be incorporated to allow an optimized schedule (with maximum resource utilization) to be achieved with satisfactory convergence.

This thesis will discuss the design and implementation of such a scheduling algorithm, intended to optimize the utilization of resources in a distributed system, in which tasks have been assigned to resources but their ordering has yet to be determined. Based on the recursive propagation messaging technique and implemented for agent systems using JADE, this system addresses the major shortcomings of its centralized and decentralized predecessors discussed in Sections 1.2 and 1.3. Furthermore, this thesis will also provide details of a sliding window approach to the same problems that modularizes the initial algorithm proposed and provides significant benefits in terms of computing time and communication bandwidth.

1.5 Thesis Outline

In Chapter 1 of this document, I have provided some basic background of distributed systems, scheduling, and some of the approaches to distributed scheduling introduced to date. Chapter 2 will describe recursive propagation in more detail and outline the major failings that Hino's method overcomes. It will also detail the sorting algorithm introduced in this work as well as the alternative sliding window approach I have developed and will provide a theoretical analysis of the new methods, explaining the major advantages that they provide.

Chapter 3 is dedicated to the implementation of the sliding window scheduling technique using JADE including the structure of a JADE interface agent and its Java scheduling partner. Using UML diagrams, these interactions between these components will also be described. Furthermore, Chapter 3 will provide a more thorough look at JADE, identifying the advantages it brings to agent system design.

Chapter 4 provides detailed results of simulations using the proposed sorting algorithm that indicate how the algorithms performance is impacted by changes to the parameters of the system. The results section then illustrates the impact of the sliding window scheduling algorithm on both computational demands and system messages required, and analyses the effect of a changing window width on system performance. Furthermore, the results compare the utilization obtained using the sliding window scheduling technique, to full optimization using no window.

Lastly, Chapter 5 provides recommendations for future research and development of this system, as well as a summary of results.

2 ALGORITHM DESIGN AND ARCHITECTURE

In this chapter, I will outline Hino's recursive propagation approach to distributed scheduling. Further, I will propose a new method for distributed scheduling that utilizes the strengths of recursive propagation, but also enhances agent intelligence to allow them to more quickly determine an optimal schedule.

2.1 Hino's Recursive Propagation

Hino, et al., developed the recursive propagation message passing technique to allow distributed scheduling agents to communicate with one another, resolve scheduling conflicts, and create feasible arrangements for tasks distributed among several resources [18][19]. To achieve this, each resource responsible for a task was given knowledge of the machine that performs the prior task of the same order and the machine that must perform the subsequent task. This knowledge constitutes the limited view of the overall system for each agent. In this section, I will outline the constraints on the scheduling problems addressed by Hino and give a detailed description of his recursive propagation method.

2.1.1 Problem Constraints

We denote each task as $\tau^{\rho,\omega}_{\alpha,\sigma}$, where the subscript α refers to the resource that performs task τ and σ to the numbered position of τ in the sequence of tasks to be performed by α, respectively. The superscripts ρ and ω respectively refer to the job and the specific numbered operation in the sequence of operations required to complete job ρ.

There are two primary constraints imposed on the scheduling problems addressed by Hino. Firstly, all tasks are subject to a precedence constraint that defines the sequence of tasks for each order. Parts are required to visit resources in a pre-determined sequence, determined by the part type. The second constraint on these problems is a resource constraint that prevents a single resource from performing two tasks at the same time, or pre-empting one task with another.

Mathematically, these two constraints restrict the possible start times that can be assigned to tasks. For each task τ, a scheduling algorithm must determine start times $t_{\alpha,\sigma}^{\rho,\omega}$ such that

$$t_{\alpha,\sigma}^{\rho,\omega+1} \geq t_{\beta,\lambda}^{\rho,\omega} + p_{\beta,\lambda}^{\rho,\omega} \qquad (1)$$

$$t_{\alpha,\sigma+1}^{\rho,\omega} \geq t_{\alpha,\sigma}^{\xi,\varsigma} + p_{\alpha,\sigma}^{\xi,\varsigma} \qquad (2)$$

relative to the start times t and processing times p, for all resources α and β, all jobs ρ and ξ, and all tasks ω and ζ. The first of these equations illustrates the precedence constraint on adjacent tasks for the same job. The second equation illustrates the constraint that no two tasks may be performed concurrently by the same resource.

2.1.2 Description of the Recursive Propagation Algorithm

Hino has published two recursive propagation algorithms [18][19]. In contrast to his first algorithm, the newer version resolves all internal conflicts before sending a message to the next agent, resulting in a large reduction in the number of messages.

To understand the recursive propagation technique, consider the schedule of tasks depicted in Figure 7(a). This representation of tasks is to be viewed as the chronological placement of operations where the time scale is horizontal and the rows of blocks represent the individual schedules of different agents. The further right the tasks are placed, the later in time they are scheduled to be performed.

Assume that agent a_1 is required to delay task $\tau^{1,1}$ (b). All operations performed by a_1 will be adjusted before the first change message is sent (c). Next, a_1 informs a_2 of the change to the first impacted task, that of order 1 (d). Agent a_2 then corrects all of its operations accordingly (e) and, since order 1 has no other operations, it sends a change message to a_1 regarding order 3.

Because task $\tau^{3,2}$ has already been adjusted, the change message from a_2 has no impact on a_1 and a result message is returned to a_2 immediately. This result is passed back to a_1 for order 1 and the next sequential operation is processed. Since there was a delay to order 2 as a result of the change to order 1, a change message is sent from a_1 to a_2 to adjust task $\tau^{2,2}$ (f). A result message is returned to a_1 and, lastly, a_1 will inform a_3 of the delay to order 3 (g).

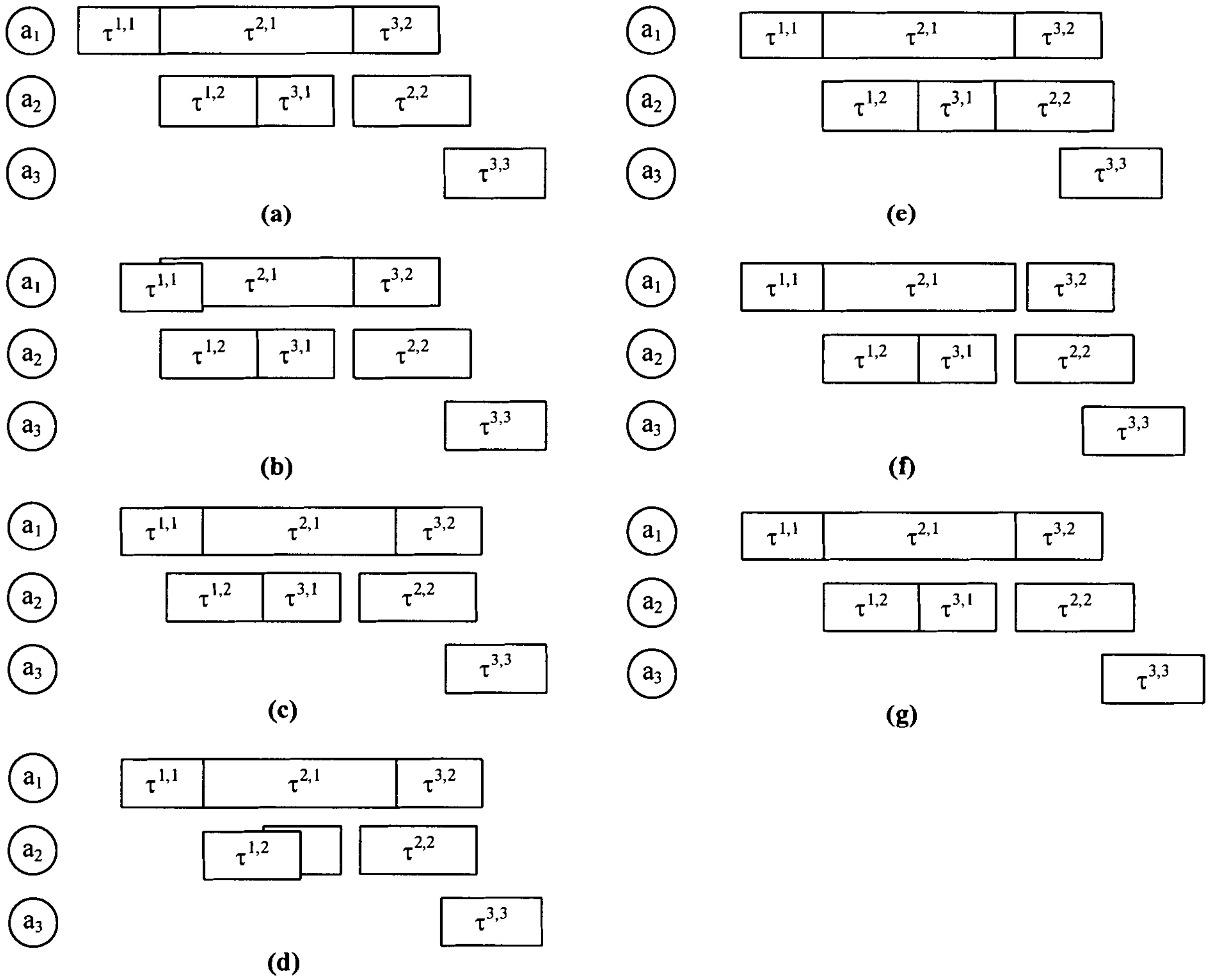


FIGURE 7. RECURSIVE PROPAGATION MESSAGE PASSING TECHNIQUE

The total number of messages required to return this schedule to feasibility using Hino's 2001 algorithm is six: three change messages and three result messages. Though the 1999 algorithm is sufficient to produce the same feasible schedule, it would have required a total of ten messages to do so. If the problem is expanded to include dozens of agents with hundreds of tasks, the savings become significant. Considering that there may be hundreds of operations that depend on the positioning of these tasks, preventing one redundant message could save hundreds or thousands of unnecessary propagations.

2.2 Enhancements to Recursive Propagation

Using recursive propagation as a foundation for agent communication, an algorithm to effectively improve resource utilization was developed.

Recursive propagation alone allows scheduling agents to resolve all precedence and resource constraint violations. However, performing tasks in the order they arrive is rarely the most efficient schedule achievable. Other orderings need to be examined, but to exhaustively evaluate every ordering of tasks would be impractical. Ten tasks on a single resource can have a total of 10! or approximately 3.6 million orderings. Furthermore, the utilization of an agent given a particular sequence of its ten tasks will be slightly different depending on the task sequence of the other agents at the instant the ordering is evaluated. So an exhaustive examination of orderings must in fact match all 3.6 million permutations with all possible orderings of the other agents in the system as well. The problem clearly explodes as systems become large.

Instead, we desired an iterative heuristic method that would approach an optimal solution as quickly as possible. Agents will choose tasks in their list and reposition them, determining from the result messages received whether the new arrangement is preferable to the first.

Jeremiah, et al., [22] studied the impact of selecting tasks based on a variety of criteria and then reassigning start times to improve the overall schedule. In his work, the operation to be rescheduled was chosen based on (among other criteria):

- Most subsequent tasks remaining to be processed for the order
- Most processing time remaining for subsequent tasks of the same order
- The ratio between the work remaining and the processing time of the current task

Unfortunately, none of these selection criteria are useful to a recursive propagation-based system, since no agents possess the type of global system knowledge on which these criteria depend.

However, based only on the limited information they do possess, agents are able to construct *active* schedules, those in which no task can be shifted any earlier in the schedule without delaying another. From this point, they can rearrange tasks to find the active schedule that results in satisfactory utilization of resources. The algorithm presented here distinguishes between two types of task sorting: sorting which fills in scheduling gaps to create an active schedule and sorting which explores active schedules to improve utilization.

2.2.1 Coarse Task Sorting

The initial feasible schedule produced by applying recursive propagation to tasks on a first-come first-served basis frequently contains a large number of gaps, large periods of idle time for each agent. Later tasks can be moved into these gaps to undoubtedly improve utilization, provided that, in doing so, the candidate task will neither violate a precedence constraint nor force other tasks to be delayed. The rearranging of tasks to fill all gaps and create active agent schedules constitutes *coarse sorting*.

To fill gaps, candidate tasks must either be the first of the order or follow a task whose completion time is early enough that the task at the end of the gap will not be pushed back by the duration of the candidate task itself. As an example, consider three tasks belonging to the same agent (Figure 8).

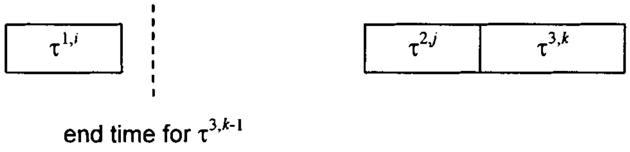


FIGURE 8. COARSE SORTING: AGENT WITH TWO TASKS

For this agent, assume that $\tau^{2,j}$ is scheduled as early as allowable by the previous agent for order 2, given the precedence constraint. The earliness of the end time of task $\tau^{3,k-1}$ allows $\tau^{3,k}$ to be exchanged with $\tau^{2,j}$ to fill the existing gap (Figure 9). Also, because the duration of the $\tau^{3,k}$ is small enough, the new positioning of this task will not delay $\tau^{2,j}$, guaranteeing a positive impact on the makespan of the orders, regardless of the number of subsequent tasks for orders 1, 2, or 3.

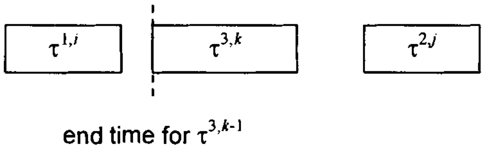


FIGURE 9. COARSE SORTING: AGENT WITH TWO TASKS (IMPROVED)

Many times, one such reordering will lead to several others. For instance, in our example above, moving task $\tau^{3,k}$ may allow $\tau^{3,k+1}$ to be moved to fill a gap as well. Agents associate new opportunities to fill gaps with the incoming change message that created the opportunity. Thus, the result returned to the initiator is the cumulative result of the time shifting of tasks and all gap filling that the change. When no gaps in the schedule are large enough for later tasks, coarse sorting is complete.

2.2.2 Fine Task Sorting

Filling all scheduling gaps doesn't necessarily imply that the schedule has been achieved the best possible utilization. There still may be tasks that, if scheduled before others, would have an overall positive impact on the system. However, it is virtually impossible for an agent to distinguish these tasks from the others in its queue based on the information it possesses. Because there is no distinction between tasks that have many subsequent operations and those with few, an agent must still be capable of reversing any decision should the result message indicate a negative impact. This trial-and-error approach to improving the schedule constitutes *fine sorting.*

As an example, consider three tasks belonging to the same agent, shown in Figure 10.

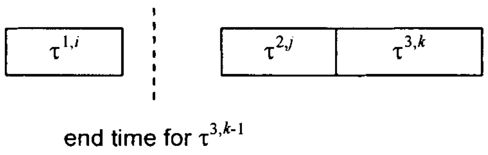


FIGURE 10. FINE SORTING: AGENT WITH TWO TASKS

For this agent, the earliness of the end time of task $\tau^{3,k-1}$ allows $\tau^{3,k}$ to be scheduled earlier, however its duration would delay task $\tau^{2,j}$ (Figure 11). This may or may not be a benefit to the overall schedule, despite the seeming benefit to this agent. If the total processing time of the tasks waiting on $\tau^{2,j}$ exceeds the processing time of the tasks waiting on $\tau^{3,k}$ by an amount greater than the improvement provided by reordering tasks $\tau^{3,k}$ and $\tau^{2,j}$, this exchange may negatively impact the overall system schedule.

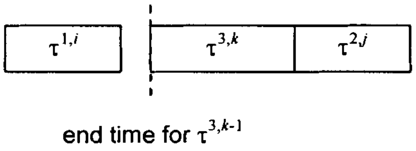


FIGURE 11. FINE SORTING: AGENT WITH TWO TASKS (AFTER FINE SORT)

Just as in the case of coarse sorting, a fine sorting exchange of tasks may also create a gap into which additional tasks can be moved. If so, these gaps will be filled with later tasks (according to the criteria described earlier) and the results attached to the original fine sorting exchange to provide a proper indication of the overall savings.

Neither coarse nor fine sorting exclusively reorders adjacent tasks in an agent's schedule. During fine sorting, tasks are moved to the earliest spot in their list based on the end time of the previous task, which may be several positions earlier. Beginning at the head of the task list, each operation is evaluated to see if there is an earlier position they might better occupy. When an operation is moved, evaluation continues for the next operation until the end of the task list is reached.

As alluded to earlier, it is possible for fine sorting to have a detrimental impact on an agent's schedule. Despite the benefit of moving a single task several positions earlier, a number of critical tasks in the middle may be delayed. When a fine sorting exchange yields negative results, the agent attempts to reverse the negative exchange incrementally. Because of the parallel nature of the scheduling process, it is impossible to guarantee that reversing an exchange will return the system to a previous state. Many other agents may have made new decisions in the time between a task exchange and its evaluation that prevent any guarantee that the reversal of a negative exchange will benefit the system. So rather than move the offending task many positions later, causing one large delay to return the sequence to its original order, the agent responds by moving the task toward the end of its list one spot at a time. As each exchange continues to produce improvements to the agent's schedule, it will continue to bubble the task outward. Once an exchange produces another negative result, it is placed in its last improved position and the next task is fine sorted.

Because fine sorting is only beneficial once coarse sorting is concluded, all agents must remain synchronized during the scheduling algorithm. First they achieve a feasible schedule using recursive propagation and then apply coarse sorting to make the most significant improvements before honing the result through fine sorting. Note that this scheduling algorithm is not deterministic. Depending on the order in which messages are processed by agents, different tasks will be eligible for coarse or fine sorting.

2.3 Sliding Window Task Scheduling

Despite the benefits of this algorithm as a distributed scheduling approach, it has its own disadvantages:

- **Message explosion.** The number of messages required to create an initial feasible schedule is significant. Even if all task conflicts have been resolved but one, the change from the remaining conflict will still require a change and result message for each subsequent task in the system, a costly requirement. Since the ordering of tasks is undoubtedly going to change through the sorting process anyway, the necessity of this initial schedule came into question.

- **Combinatorial complexity.** The problem of determining the optimal ordering of tasks, as mentioned early, is combinatorial in nature. Despite the savings of a heuristic approach in this regard, further savings were desired.

- **Lack of responsiveness to dynamic changes**. In spite of the algorithm's scalability, the addition of a new resource or a new order still requires the reapplication of the scheduling procedure to provide the optimal integration of the new order. This meant that the current technique was not a truly dynamic approach.

To combat each of these deficiencies, the sliding window method was designed [24][25]. The sliding window scheduling technique is based upon breaking a large problem into smaller more manageable sub-problems. By maximizing utilization for each of the sub-problems, we will necessarily have maximized utilization for the original scheduling problem.

Consider a problem in which agents have twelve tasks each to sort. Now consider that, instead of attempting to sort the entire list of twelve, each agent focused on a window of time in which only four of these tasks would fit. When the arrangement of the first four tasks was satisfactory (utilization had been maximized), the window would be advanced and the next four tasks examined. Messages are no longer propagated for tasks beyond the end of the window, since the implications of these messages have no effect on the current set of tasks.

This system addresses all three shortcomings of the original method:

- **Message explosion.** By limiting the messages to the window of focus, the message savings are two-fold. First, the messages of initial scheduling are reduced, since conflicts beyond the current window are only addressed as the window advances. Second, messages during sorting are also propagated only as far as the current window, producing further savings in communication overhead.

- **Combinatorial complexity.** In the problem described above, the number of permutations for each agent to evaluate in an exhaustive full optimization would be 12! or approximately 479 million. By focusing on reduced windows, each of the three sub-problems requires only 4! or 24 orderings to be examined, 72 total.

- **Responsiveness to dynamic changes**. Because the tasks are only scheduled one window at a time, tasks in completed windows can be processed while the resource is continuing to determine the rest of its schedule. Meanwhile, if new orders are added or new agents come on-line, their tasks can be incorporated into the current window and future windows without impacting previously scheduled tasks, allowing for scheduling on the fly. This makes the algorithm completely adaptable to a changing work environment.

The sliding window approach limits the focus of the agents to tasks within a specified time window beginning with time t_1 and extending to, but not including, time t_2, denoted $[t_1,t_2)$. The resulting schedule is, at any instant, only feasible for tasks beginning before t_2. Consider, the detailed example of the sliding window technique using recursive propagation illustrated in Figure 12.

Figure 12(a) illustrates the feasible schedule that three agents a_1, a_2, and a_3 would achieve for the window $[t_1, t_2)$ given an example task distribution. Messages for jobs 1 and 2 would be propagated fully to resolve all conflicts. But job 3 would not be scheduled in the current window, since the start time of $\tau^{3,1}$ exceeds t_2. The three tasks of job 3 will be left in violation of their precedence constraints ($\tau^{3,2}$ begins before $\tau^{3,1}$ has concluded) until the window advances or until they are reordered.

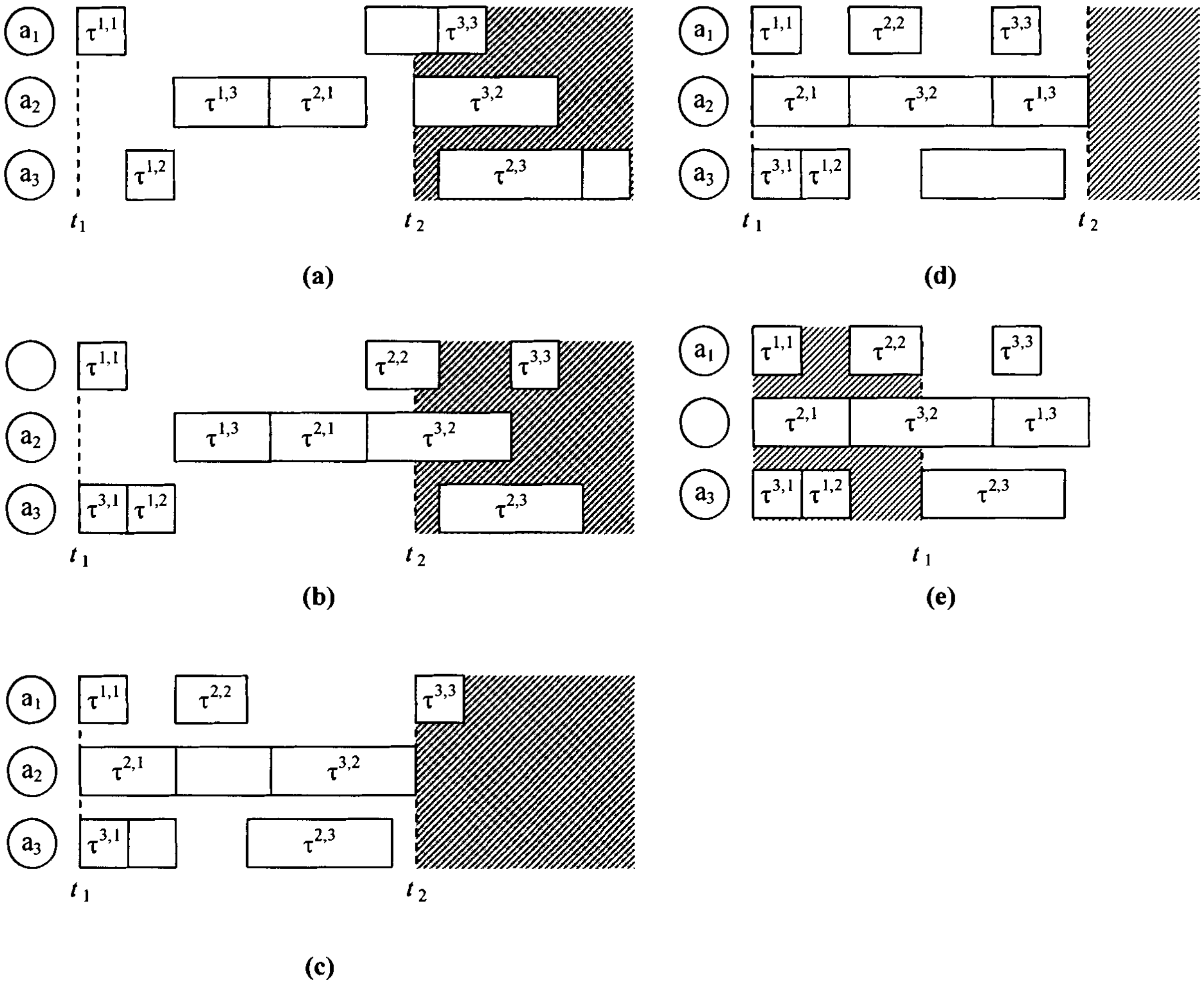


FIGURE 12. SLIDING WINDOW EXAMPLE

In Figure 12(b), agent a_3 reorders its tasks so that $\tau^{3,1}$ is performed before $\tau^{1,2}$. A message will be sent to update $\tau^{3,2}$ and, since it now begins in the current window, task $\tau^{3,3}$ will be updated as well. In (c), $\tau^{2,1}$ is similarly reordered by agent a_2 and its subsequent tasks moved earlier as a result. At this point, the agents can attempt to fine sort additional tasks. For example, agent a_2 could exchange tasks $\tau^{1,3}$ and $\tau^{3,2}$ as in (d).

Once all task conflicts within the window have been resolved and sorting is complete, the agents agree to advance the window by changing the values of t_1 and t_2 (e). The step size of this advancement, Γ, can be varied between any value greater than zero and any value no larger than the window size, Δ. A step smaller than Δ will potentially allow the fine sorting of some tasks to be revisited, which may improve the schedule given the changing configurations of neighboring agents. However, a step larger than Δ will likely leave tasks unresolved between windows.

2.4 System Architecture

2.4.1 Structure of Tasks and Task Lists

The principal objects manipulated during sliding window scheduling are the tasks assigned to each agent in the system. Each task is associated with a specific order, a part that has to be built or a sequence of operations that must be performed. As well, each task has a defined processing time. These two parameters are assumed to be fixed properties. But there are several other components of the task class that change throughout the scheduling process.

The structure of the Task class is shown below in Figure 13. The `order` field refers to a unique integer identifying each job. Operations belonging to a sequence for the same job will have the same order number. The `duration` field indicates the processing time for the task. Both the order and duration fields of the task class are provided when the task is created. The `start` time is updated constantly throughout the scheduling process.

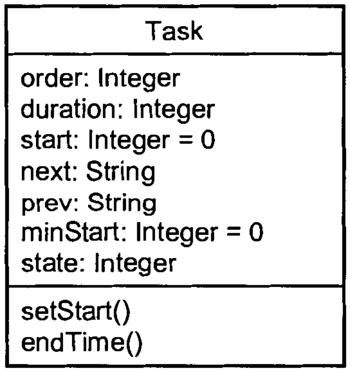


FIGURE 13. TASK CLASS STRUCTURE

The next and prev variables hold the names of the agents that perform the task immediately before and immediately after the current operation for the current job. For the first task of each job, the prev field will hold NULL; for the last task of each job, the next field will hold NULL. Messages for propagating changes will be sent to the addresses contained in these fields.

This sorting algorithm relies heavily on the end times of previous tasks. Though each agent could request this information from the previous resource for a given order, the number of messages required would be extensive, since information would be requested not only for each task exchanged but for each task considered for exchange. Instead, information from the previous message received is kept with each associated task in a field called the minStart. This value indicates the earliest start time allowable by the previous agent without violating precedence constraints.

Because messages are only propagated for tasks inside the window, some tasks beyond the window could potentially exist with outdated minStart information. When agents make sorting decisions based on outdated information, instabilities occur and the potential exists for the scheduling algorithm to diverge. To eliminate this problem, each task is assigned a `state` that indicates whether its `minStart` information is up-to-date and whether exchanges can be made for the current task. No task is reordered unless it has been deemed SAFE (its information is current). Likewise, no agent considers its window complete until all tasks within it are SAFE. The state field changes the recursive propagation algorithm slightly, in that not only are changes in start times propagated, but also changes in state value. Until this criterion was included, agents would occasionally advance the window with unresolved tasks still inside.

There are five possible states for each task in an agent's task list as illustrated by Figure 8: UNSAFE, SAFE, TELLS, TELLP, and TELLU. The first and second are relatively straightforward. Each task, apart from the first task for each order is assumed UNSAFE (with outdated `minStart` information) until it is updated by a change message from the preceding agent. When a message indicates that it has the most recent information available, the state becomes SAFE. Both UNSAFE and SAFE tasks require no further propagation of information.

The remaining three state values indicate that information needs to be propagated to the next agent for the current order. TELLS is used for tasks that are SAFE and need to propagate change information to the next agent for the current job. This state exists for tasks that are inside the current window.

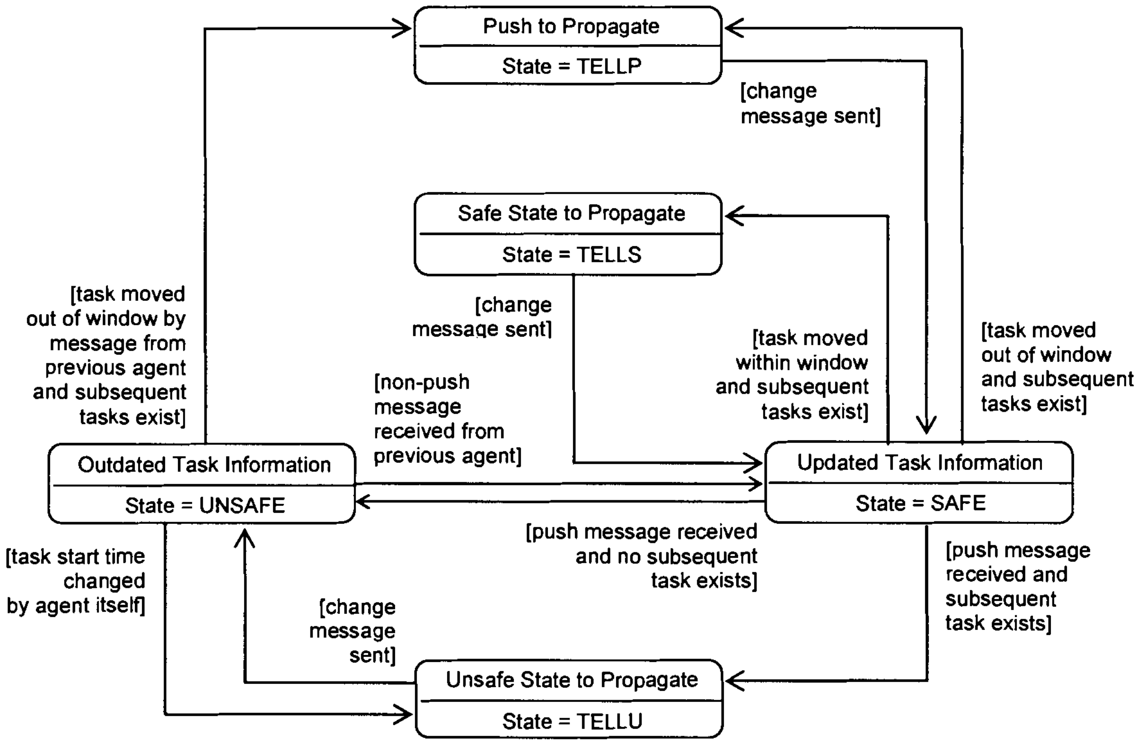

Push to Propagate
State = TELLP
[change message sent]
Safe State to Propagate
State = TELLS
[task moved out of window by message from previous agent and subsequent tasks exist]
[change message sent]
[non-push message received from previous agent]
[task moved within window and subsequent tasks exist]
[task moved out of window and subsequent tasks exist]
Outdated Task Information
State = UNSAFE
Updated Task Information
State = SAFE
[push message received and no subsequent task exists]
[task start time changed by agent itself]
[change message sent]
[push message received and subsequent task exists]
Unsafe State to Propagate
State = TELLU

FIGURE 14. TASK STATE DIAGRAM

When tasks are pushed out of the current window by another SAFE task, they will no longer propagate any future changes to the agents that follow, despite possibly having up-to-date information themselves. This condition is called a *push*. If the current task has been pushed out of the window, but its information is current, its state will be TELLP, and it will inform the next task that it should consider itself UNSAFE until it receives new information. Any task that is told to become UNSAFE will change state to TELLU if it has further tasks to inform. Throughout sliding window scheduling, task states will continue to change between SAFE, UNSAFE, and all forms of the tell states as they are moved in and out of the scheduling window. Section 2.4.2 illustrates an example.

Each agent possesses a list of the tasks that it is required to perform. Elements of this list are continually reordered and the impacts of these actions monitored until the agent is satisfied that it has best possible arrangement of tasks. Additionally, elements can be added to this list on-the-fly as agents are scheduling. Because of the flexibility required, a linked list is used to implement the task list of each agent. The primary functions of the task list are `push`, `pop`, and `moveBefore`. The first two add and subtract tasks from the list. The last is used during coarse and fine sorting to reorder the task nodes.

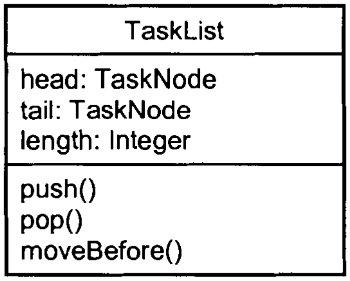


FIGURE 15. TASK LIST CLASS STRUCTURE

2.4.2 Structure of Messages and Message Lists

Several types of messages are required for system operation. The most crucial are the change and report messages implemented for recursive propagation of scheduling changes, but other message types are required for synchronization (Table 1).

As mentioned earlier, all agents remain synchronized throughout scheduling. So, whether using the window approach or not, the agents will simultaneously be resolving precedence constraint violations to create a feasible initial schedule, followed by coarse sorting, fine sorting, and the handling of all remaining messages before concluding.

Between each of these phases of operation, there is an exchange of synchronization messages required. In the case of the sliding window approach, the window will then be advanced and the agents will proceed to create a feasible schedule for the next window of tasks.

TABLE 1. MESSAGE TYPES

TYPE	VALUE	DESCRIPTION
Add (ADD)	1	Add a new task to the agent's TaskList
Change (CHG)	2	Task needs to update its start time and/or minstart
Result (RES)	3	Cumulative result of changes
Ready (RDY)	4	Agent is satisfied with the current schedule
Reply (RPY)	5	Ready agent has received ready messages from all peers

The reasons for synchronization are several. First, without generating a feasible schedule for at least a segment of the tasks, reordering of tasks can lead to instabilities. Second, because of the large gains resulting from coarse sorting, agents who attempt to fine sort tasks before their peers have completed coarse sorting may be making unnecessary effort. Finally, all agents must schedule the same sized window and be synchronized in doing so for the sliding window algorithm to be effective, since agents will not propagate messages beyond their own window despite the fact that other agents may have other windows of focus.

But, unlike multi-agent systems that use broadcast messages, the sliding window scheduling algorithm still uses targeted message recipients for synchronization. Since agents are only aware of the agents who perform tasks immediately following and immediately preceding their own, they do not necessarily have knowledge of all agents on the system. In fact, in the case that an agent performs single-task orders, it may not be aware of any other agents. All agents who perform adjacent tasks to those being scheduled by an agent constitute the *peers* of that agent.

As in the case of a single agent isolated from the others, the possibility exists for multiple subsystems of agents to schedule their jobs independently of one another, as shown in Figure 16 (a), where lines between agents a_i indicate a peer relationship. But with even a single common agent (b), these systems will remain synchronized. For example, despite the fact that agents a_1 and a_5 in (b) are not peers based on their tasks, they will be kept in synch due to their mutual dependence on peer a_4.

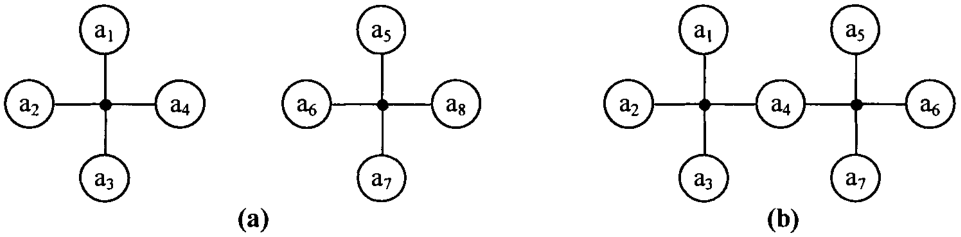


FIGURE 16. AGENT PEER RELATIONSHIPS

Each agent sends a *ready* message to its peers when it is satisfied with the current scheduling window during each stage of scheduling. Because of the reciprocal nature of the peer relationship, each agent will also receive a ready message from each of its peers.

Despite synchronization messages, agents will still advance their windows independently, once a ready message has been received from all peers and the agent itself is satisfied with the current schedule. But the ready messages themselves do not ensure that all agents stay synchronized, particularly in light of the fact that some peers could be separated by several degrees. To combat this phenomenon, the *reply* message type was created. Once all peers are ready and the agent is ready to move on, reply messages are sent out to confirm that all peers are on the same page. In order to proceed to the next phase of scheduling, an agent must have received not only a ready message from all its peers, but also a reply message for each ready message it sent.

Below, Figure 17 outlines the message structure used by scheduling agents.

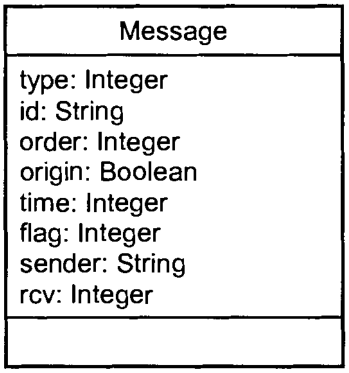


FIGURE 17. MESSAGE CLASS STRUCTURE

Message `type` values include change, result, and ready and reply. For change and result messages, the `id` field is used to distinguish the changes responsible for any given impact on the schedule (since multiple concurrent changes may affect the same operation). The `order` field indicates the order number that is currently being affected. Since no agent will perform two tasks for the same order, the order number will uniquely identify the task in question. The `origin` field indicates whether the message was self-generated or received from another agent and is used to set the task `minStart` field. The `time` field indicates the earliest start time of the indicated operation for change messages and the overall delay/improvement to the system for result messages.

The message `flag` is used to propagate state values to other agents. Typically, messages are only passed on when they have forced a change to the current task. These messages use a CLEAR flag and are handled according to `the` original recursive propagation technique. To ensure that each new window is filled only with updated tasks, initial feasible scheduling begins by propagating messages with FORCE flags for all SAFE tasks within the window. The FORCE flag ensures that a message is propagated until the end of the window is reached, regardless of its impact on the recipient. For UNSAFE tasks and those pushed out of the current window, a PUSH message flag is used to force the recipient back to an UNSAFE state.

As an example, consider a change to agent a_1 that delays $\tau^{1,1}$ toward the end of the current scheduling window (Figure 18). In (b), the task state will be set to TELLS to remind the agent to inform a_2 of the change. A change message with a CLEAR flag will be received by a_2 and task $\tau^{1,2}$ will be pushed out of the current window. As a result, any future change to this task will only be propagated if it brings $\tau^{1,2}$ back into the current window. No further delays, regardless of how substantial, will be passed on to other agents. To make sure that future agents are aware that they will no longer be updated, the task state is set to TELLP (c) and a change message is sent to a₃ with a PUSH flag. If there are subsequent tasks to notify, the state of $\tau^{1,3}$ will be set to TELLU and a further PUSH message sent. Otherwise, the state will be set to UNSAFE and results will be returned. The `rcv` and `sender` fields were added to the message structure for JADE implementation and will be discussed in Chapter 3.

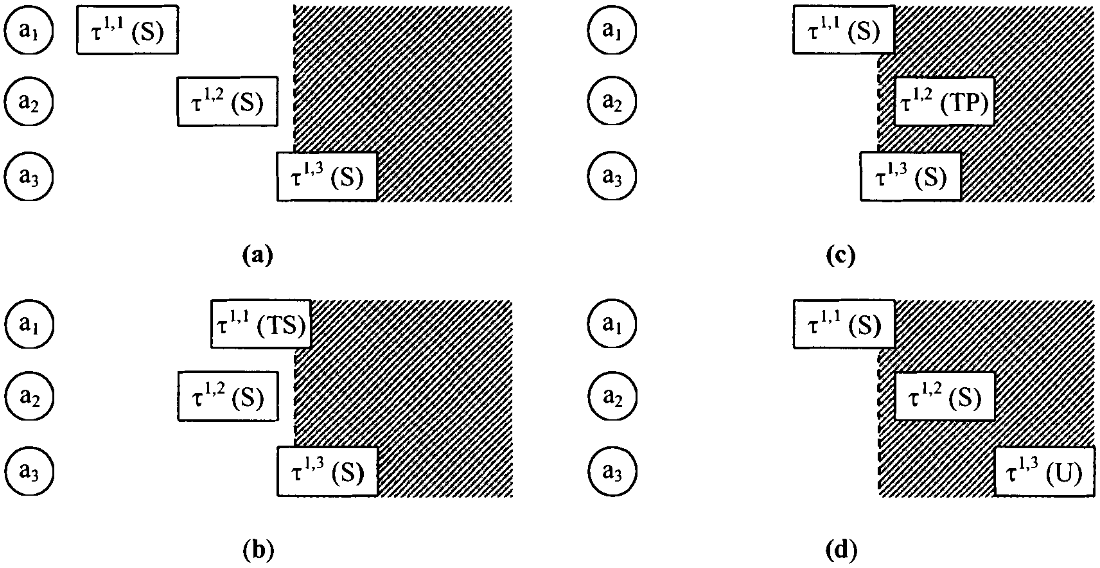


FIGURE 18. TASK STATES AND MESSAGES FLAGS

2.4.3 Detailed Description of the Sliding Window Scheduling Algorithm

As mentioned, the first step each agent takes in scheduling is to send a FORCE change message for each SAFE process in the current window. There may be redundancies here, in that already SAFE operations may receive messages confirming their original start time, but the cost of a few extra messages is worth the risks they eliminate. The first UNSAFE task and all tasks that follow are pushed beyond the current window. This operation frees up space in the current window for any of the SAFE tasks that the agent may have placed later in its task sequence.

This precaution explains the reason task $\tau^{3,2}$ in Figure 12 (a) did not immediately follow task $\tau^{2,1}$. Having never been updated, it would be recognized as UNSAFE and pushed to the windows edge. Any SAFE tasks that are pushed out as a result of this step will most likely be reordered back inside the window during the coarse or fine sorting phase.

Following this initial scheduling phase, ready messages are sent to each of the agent peers and the agent enters a waiting state, still processing change messages from other agents, but waiting until all ready and reply messages have arrived before proceeding to the sorting state.

Once the agents are synchronized, the tasks are coarsely sorted for the current window until all gaps in the schedule between t_1 and t_2 (the start and end times of the window). Only SAFE tasks whose `minStart` values allow them to be brought inside the window to fill these gaps are selected and no gaps beyond the window are addressed. When no gaps exist within the window that can be filled by SAFE tasks, the agent synchronizes with its peers, awaiting messages in return before moving on.

When all ready and reply messages have been processed, fine sorting begins. Again, only tasks within the window are selected to be exchanged. They are never moved earlier than t_1 based on the assumption that tasks scheduled previously are already being processed by the current resource. Any gaps created in the current window by the fine sorting of tasks are filled by later tasks using the coarse sorting approach. A pseudo code representation of the sorting logic appears below.

```
if (state == COARSE) { coarseSort(); }
else if (insideWindow(currentTask)) {
if (!acceptLastChange()) { undoChange(); }
else if ((state == SCHED) && (isSafe(currentTask))
   { notifyNext(); }
else if (state == FINE) { fineSort(); }
   ++currentTask;
} else {
   sendMsg(Ready);
}
```

When fine sorting changes are revealed to be detrimental, the offending task is bubbled toward the end of the list, in an attempt to restore a favourable sequence. Agents will synchronize when fine sorting has completed and wait for all ready and reply messages before continuing. Once all peers have replied and the agent wishes to advance, the remaining messages in the incoming message queue are handled, the agents again synchronize, and then each advances its window.

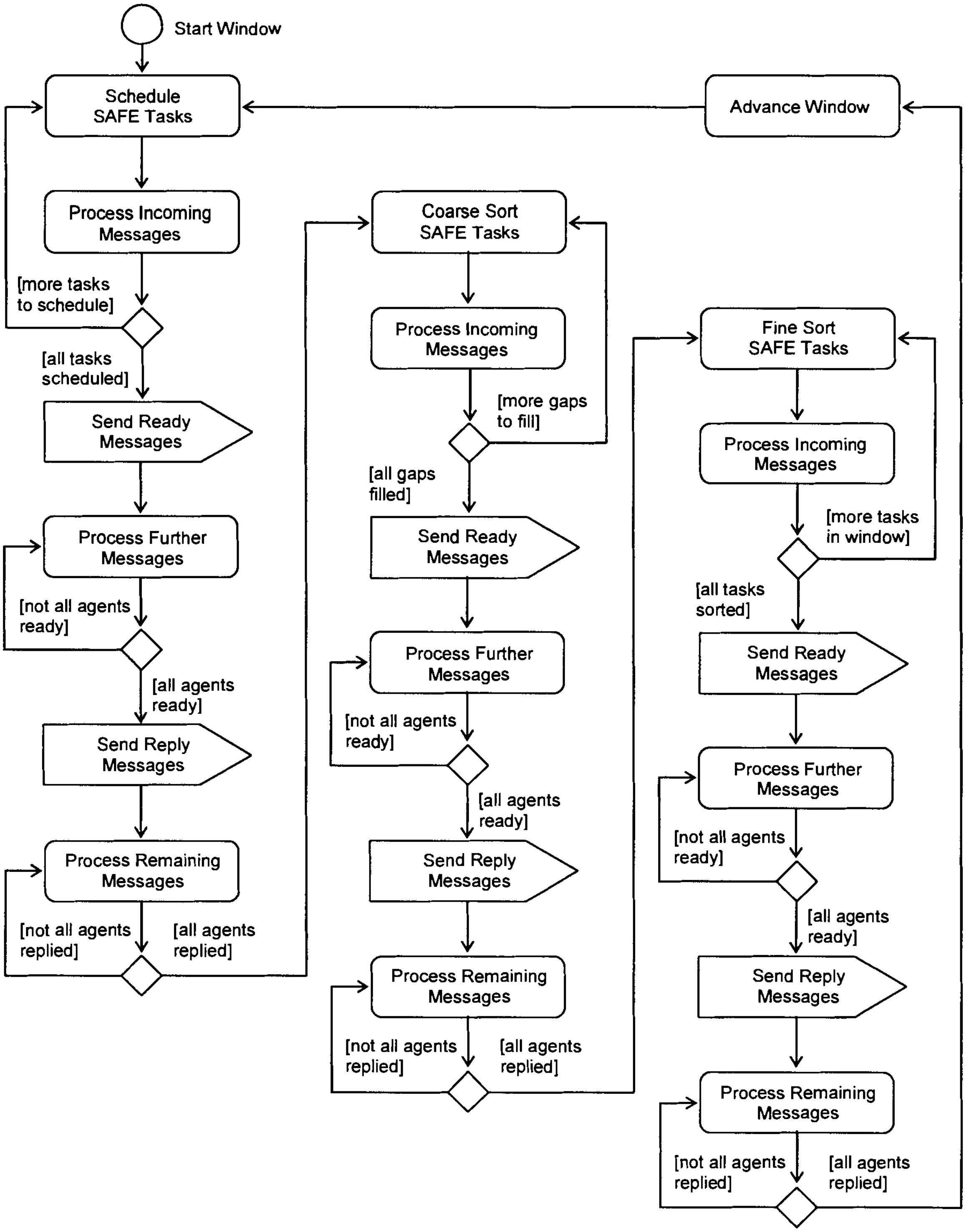
Below is a pseudo code representation of the message-handling logic.

```
if (newMsgArrived) {
   if (messageType == Ready) { handleReady(); }
   else if (messageType == Reply) { handleReply(); }
   else if (messageType == Change) {
      if (!coarseSort()) { resolveConflicts(); }
      if (change) { notifyNext(); }
      else { reportToLast(); } }
   else if (messageType == Result) {
      if (!coarseSort()) {
         if (moreToNotify) { notifyNext(); }
         else { reportToLast(); }
      }
   }
} else { waitForMessage(); }
```

The `handleReady` and `handleReply` functions allow the agent to keep track of peers from which it is still waiting for readiness confirmation. For each change or result message, the agent will determine whether a new opportunity to coarsely sort tasks has been created and verify that the state of the agent is either COARSE or FINE before filling any gaps. If no such opportunity exists or the agent state prevents it, the `resolveConflicts` procedure follows the logical steps of Hino's recursive propagation in adjusting the start times of tasks based on the incoming message. Once changes have been made to tasks, the `notifyNext` or `reportToLast` function is called accordingly.

The corresponding agent activity diagram and agent state diagram are shown in Figures 19 and 20, respectively. Pseudo code of the state transitions are presented below

```
switch (state) {
      case: SCHED
            if (allTasksScheduled()) { state = WAITS; }
break;
      case: WAITS
            if (allAgentsReady()) { state = COARSE; }
break;
      case: COARSE
            if (scheduleActive()) { state = WAITC; } break;
      case: WAITC
            if (allAgentsReady()) { state = FINE; } break;
      case: FINE
            if (scheduleSatisfactory()) { state = WAITF; }
break;
      case: WAITF
            if (allAgentsReady()) { state = EMPTY; } break;
      case: EMPTY
            if (messageQueue.length==0) { state = WAITE; }
break;
      case: WAITE
            if (allAgentsReady()) {
         advanceWindow(DELTA);
         state = SCHED;
      } break;
```



}

FIGURE 19. AGENT ACTIVITY DIAGRAM

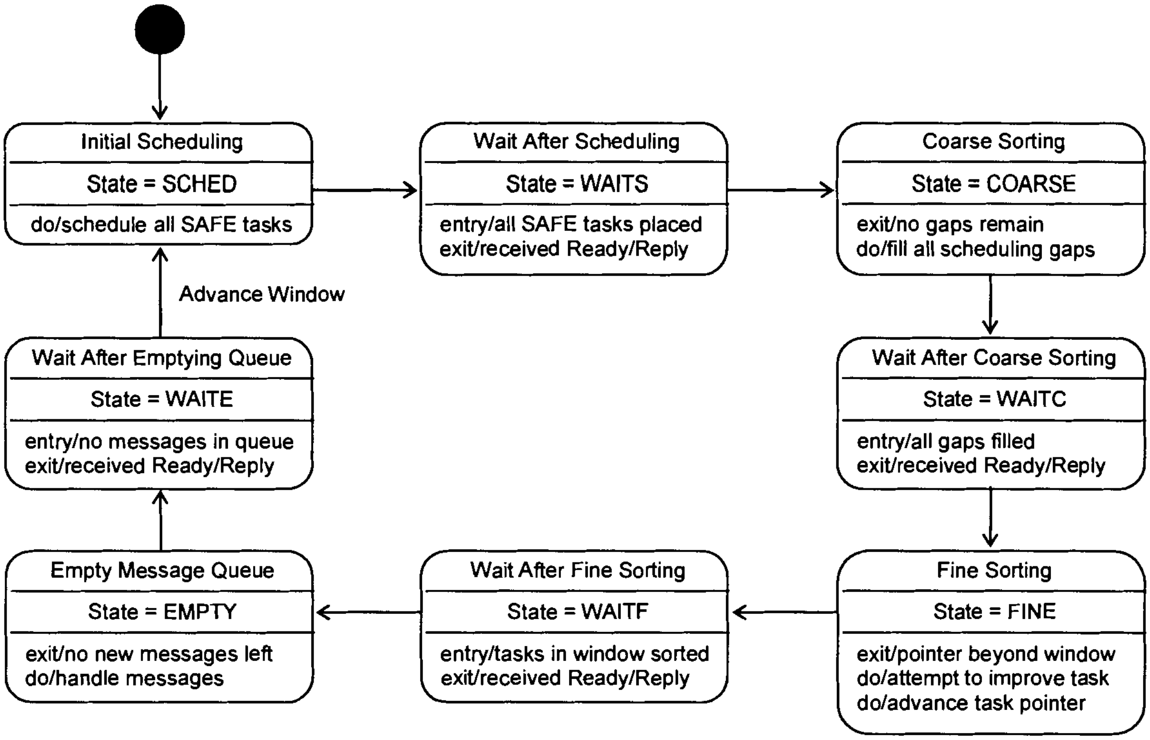

Initial Scheduling
State = SCHED
do/schedule all SAFE tasks
Wait After Scheduling
State = WAITS
entry/all SAFE tasks placed
exit/received Ready/Reply
Coarse Sorting
State = COARSE
exit/no gaps remain
do/fill all scheduling gaps
Advance Window
Wait After Emptying Queue
State = WAITE
entry/no messages in queue
exit/received Ready/Reply
Wait After Coarse Sorting
State = WAITC
entry/all gaps filled
exit/received Ready/Reply
Empty Message Queue
State = EMPTY
exit/no new messages left
do/handle messages
Wait After Fine Sorting
State = WAITF
entry/tasks in window sorted
exit/received Ready/Reply
Fine Sorting
State = FINE
exit/pointer beyond window
do/attempt to improve task
do/advance task pointer

FIGURE 20. AGENT STATE DIAGRAM

In this chapter, the algorithm used to schedule sequenced tasks has been outlined. This algorithm relies on recursive propagation to provide distributed communication while simultaneously distributing the decision-making required among the agents in the system. As we shall see in Chapter 4, not only does the sorting technique itself provide improvements in performance when compared with Hino's exhaustive searches, the sliding window approach provides further improvements. In the next chapter, we examine the implementation of this system using Java and JADE and the design of agents.

3 IMPLEMENTATION

In this chapter, I will identify more specifically the structure of agents in the proposed scheduling system design and the implementation of these agents in Java for JADE agents systems.

3.1 Introduction to JADE

The Java Agent Development Framework (JADE) is middleware designed to facilitate the development of multi-agent peer-to-peer applications. Developed by Telecom Italia Labs in Italy, the software has been shared as open source since February 2000. JADE is designed using Java, providing interoperability between agents running on varied operating systems, and can be used with any number of versions of Java for both fixed and mobile devices. Because of this feature and its small footprint, JADE agents can run everywhere from powerful workstations to mobile cellular phones.

JADE allows agents to cooperate and pass messages using FIPA-compliant message structures and a simple set of API routines. In a JADE agent system, agents are able to register themselves and the services that they can provide (i.e. machine functions, in the context of our scheduling system) with a *directory facilitator* service, which then allows all agents to look up peers according to the services they provide. The directory facilitator also ensures that each agent is assigned a unique agent identifier (AID) that allows it to be located and identified as a message recipient.

The message protocol utilized by JADE is the agent communication language (ACL) message structure (Figure 21).

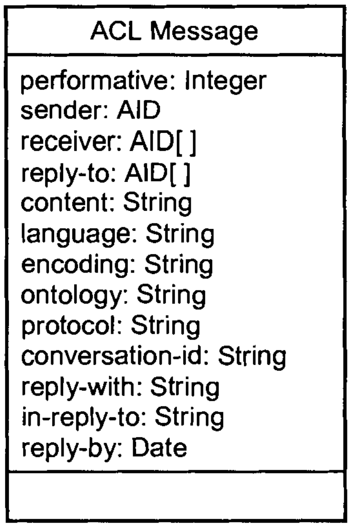


FIGURE 21. ACL MESSAGE STRUCTURE

Of the fields that constitute the ACL message type, several are of use to sliding window scheduling agents:

- **sender**: the agent from whom the message is being sent
- **receiver**: the agent to whom the message is intended
- **content**: the substance of the message

The ACL message structure provides the opportunity for complex communication between agents, where agents negotiate with multiple peers using a variety of languages and message encoding techniques, and indicating their intentions with the inherent message performatives provided by ACL. However, the other fields, while applicable to other multi-agent system implementations, are not required by the scheduling agents, primarily because of the message substructure we have already developed for communication between agents.

For JADE versions prior to 3.2, JADE agents were purely single-threaded entities, though multiple behaviours could be implemented to share the processing time on this single thread for multi-tasking agents. JADE 3.2 introduced the `ThreadedBehaviourFactory`, which permits true multi-threading. Using version 3.2, blocking socket commands in Java such as `accept` and `read` can be used to suspend a single behaviour without freezing the entire agent.

Each instantiation of the JADE run-time is called a container. While multiple containers can exist on the same the platform, there can be only a single main container on which the directory facilitator resides. Communication between agents on different platforms requires additional message transfer protocols to be used, namely HTTP.

The system developed for this thesis was implemented using JADE version 3.2 and the Java 2 SDK version 1.5.

3.1.1 Building Scheduling Agents on Top of JADE

Whether JADE is the optimal framework for developing distributed applications remains to be seen. Research by Ng, et al., [27] and Chen, et. al, [28] have investigated the communication and processing overhead that JADE systems inherit as compared to other available distributed environments. For the purpose of this thesis, JADE was chosen to verify the successful implementation of the scheduling algorithm on distributed resources.

Because JADE may not be the optimal environment for our agent system, the scheduling agent class itself was not developed in JADE but rather in pure Java. A JADE entity provides the communication and registration benefits of the JADE framework, but none of the decision-making required by the sliding window algorithm. The interface between these two elements is handled by sockets and a simple send command, which delivers a String (a representation of the message structure of Figure 17, with delineators to separate fields) and an address to which it is to be sent. These messages are then passed between JADE agents across some form of network (wireline, wireless, peer-to-peer, or otherwise), as shown in Figure 22.

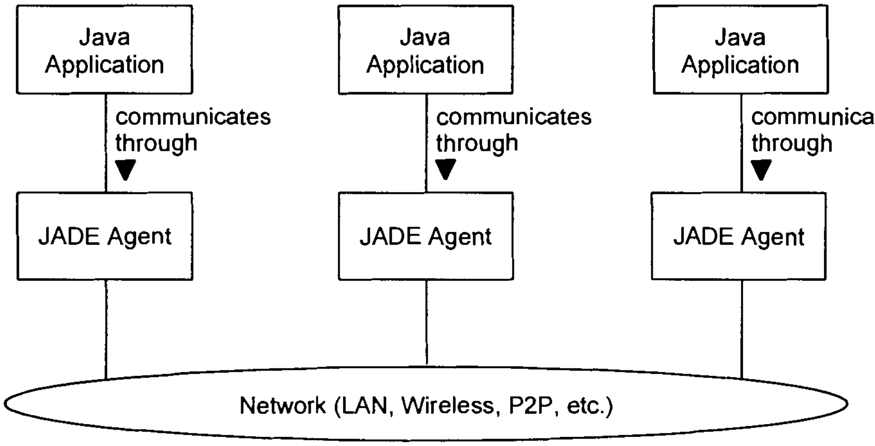


FIGURE 22. NETWORKED SCHEDULING AGENTS

Provided that the interface between the Java scheduling agent class and the middleware agent platform remained the same, this simple API allows the JADE infrastructure to be replaced by a more suitable alternative platform, should JADE prove insufficient for future generations of the scheduling system. With this implementation, and the abstraction of the middleware from the scheduling agent itself, the scheduling agent software has the potential to be ported to workstations running FIPA-OS, cell phones running JADE, and PDA's using other multi-agent frameworks without redesign or even recompilation.

3.2 Composite Scheduling Entity

As mentioned, for each scheduling agent implemented in pure Java, a JADE agent is created to provide a presence on the JADE platform and handle communication with other agents. Together these two paired classes constitute a *composite scheduling entity* (CSE), both equally vital to the successful scheduling of a single resource. Though both written in Java, they will be distinguished hereafter as the Java scheduling agent (the class written in pure Java) and the JADE interface agent (the class written using JADE libraries).

3.2.1 Implementation of the JADE Interface Agent Class

The JADE interface agent class is designed as an agent with two behaviours: one that handles incoming ACL messages and a second that takes information from its partner Java scheduling agent and packages it in ACL messages destined for other agents on the system. Each JADE interface agent is instantiated with the port numbers it will use to connect to its partner Java scheduling agent. It is also given a name, which is then shared by both components of the CSE. On start up, the JADE interface agent registers itself with the directory facilitator, opens a ServerSocket that will await commands from the Java scheduling agent, and sends via Socket a message to inform its partner of their shared name before cycling through its two behaviours: JADEToJavaBehaviour and JavaToJADEBehaviour. These two behaviours operate concurrently on separate threads.

The JADEToJavaBehaviour is designed to handle incoming messages from other JADE interface agents on the platform. When a message arrives, it is automatically placed into the incoming message queue of the JADE interface agent by the underlying JADE framework. For each message found in this queue, the JADE interface agent sends the content field of the received ACLMessage to the Java scheduling agent over their dedicated Socket. Once this socket is opened, is sent across the socket. The JADEToJavaBehaviour is then blocked, awaiting a new ACL message before it will be executed again.

The JavaToJADEBehaviour works in an opposite fashion. Upon start up, a ServerSocket is created and the accept Java function is used to wait for a client connection. The only client that will ever connect with the JADE interface agent in this way is its partner Java scheduling agent. Once the socket is opened, it will be blocked by a call to readLine. The Java scheduling agent will eventually pass it two strings: the intended message recipient (the name of its scheduling agent peer) and the content of the message (a String representation of the structured Message). Once the JADE interface agent has received these two strings, it constructs a new ACL message, filling in the receiver and content fields appropriately and uses the JADE `send` command to deliver the message using JADE. The behaviour then listens on the socket for a new client connection (a new message to send).

The JADE interface agent class has no intelligence or logic built in to handle incoming messages in a particular way. It simply passes messages to the Java scheduler for processing.

3.2.2 Implementation of the Java Scheduling Agent Class

The Java scheduling agent is, as expected, the more complex component of the two elements of the CSE, not only due to the decision-making it performs but also in its handling of incoming messages from the JADE interface agent.

The Java scheduling agent also has two primary behaviours. These operate on separate threads. The `SchedulingBehaviour` performs all necessary functions associated with the sliding window scheduling algorithm. It processes change, result, ready, and reply messages as they are added to the `myMsgs` MessageList and sends messages to the JADE interface agent (using sockets) when it needs to communicate with another CSE. Incoming change messages are added to a MessageList called `mySched` before changes are propagated. Each change message in `mySched` must receive a corresponding result before it can be discarded. Also, using a tellList, a hearList, a replyList, and a delayList, the agent records all peers from which it still expects ready or reply messages and all peers to which it owes ready and reply messages in return.

On a second thread, a MessagingBehaviour communicates with the JavaToJADEBehaviour of the JADE interface agent. On start up, a ServerSocket is created that listens for connection from the JADE interface agent. When the JADE interface agent connects, a readLine call blocks the thread until a new ACL message arrives. The content field is read as a single String that is then converted into a Message and added to the internal message queue of the Java scheduling agent, where it is read and processed by the SchedulingBehaviour.

The MessagingBehaviour also ensures that received messages are handled in sequence. Since network delays can cause JADE messages to arrive out of order, there can be problems in agent performance, particularly if messages changing task states are mixed up. The sender and rcv fields of the Message structure are used to maintain the sequence of incoming messages. If an unexpected rcv number is received from a particular sender, the MessagingBehaviour adds the early message to the holdList until the expected message arrives.

The class structure of a composite scheduling entity is shown in Figure 23.

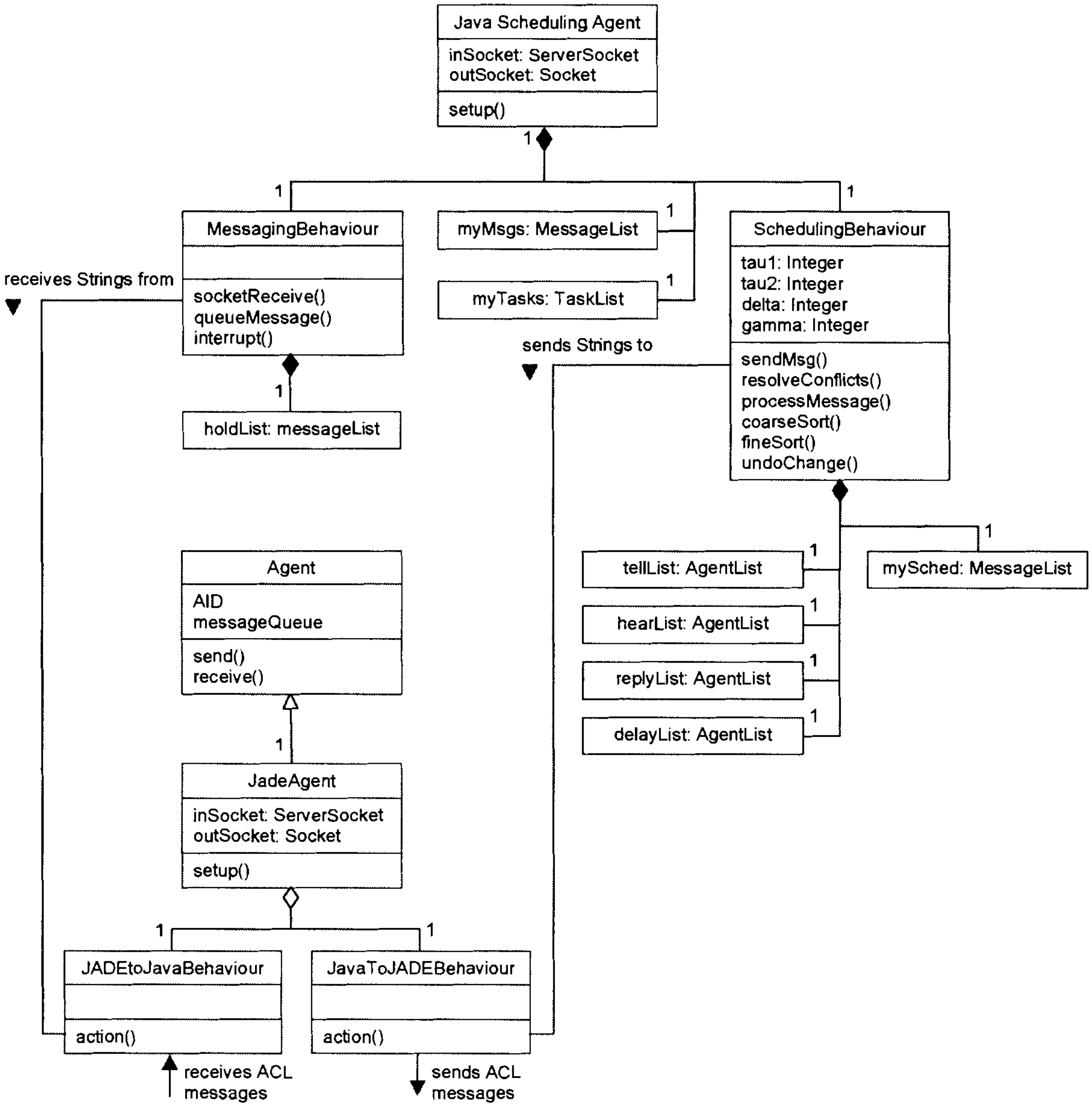


FIGURE 23. COMPOSITE SCHEDULING ENTITY CLASS DIAGRAM

The following table summarizes all the primitive class structures used to implement the sliding window scheduling algorithm including all public properties and methods.

TABLE 2. CLASS STRUCTURES OF SCHEDULER IMPLEMENTATION

Class	Member	Description
AgentList	head (AgentNode)	Pointer to the head of the linked list
	tail (AgentNode)	Pointer to the tail of the linked list
	locked (Boolean)	Protects thread-safe regions of code
	length (Integer)	Length of the linked list
AgentNode	next (AgentNode)	Pointer to the next node of a linked list
	last (AgentNode)	Pointer to the previous node of a linked list
	myAgent (String)	Name of the agent represented by the node
MessageList	head (MessageNode)	Pointer to the head of the linked list
	tail (MessageNode)	Pointer to the tail of the linked list
	locked (Boolean)	Protects thread-safe regions of code
	length (Integer)	Length of the linked list
MessageNode	next (MessageNode)	Pointer to the next node of a linked list
	last (MessageNode)	Pointer to the previous node of a linked list
	myMessage (Message)	Name of the agent represented by the node
Message	type (Integer)	Message type
	id (String)	Unique identifier for a chain of messages
	job (Integer)	Job for which the message is intended
	origin (Boolean)	Is the message internally generated?
	time (Integer)	New start time or result passed
	flag (Integer)	Clear, Force, or Push
	rcv (Integer)	Sequences incoming messages
	sender (String)	Name of the message sender
TaskList	head (TaskNode)	Pointer to the head of the linked list
	tail (TaskNode)	Pointer to the tail of the linked list
	length (Integer)	Length of the linked list
TaskNode	next (TaskNode)	Pointer to the next node of a linked list
	last (TaskNode)	Pointer to the previous node of a linked list
	state (Integer)	Safe, Unsafe, TellS, TellU, TellP
	myTask (Task)	Name of the agent represented by the node
Task	job (Integer)	Job to which the task belongs
	start (Integer)	Start time of the task
	duration (Integer)	Processing time of the task
	minStart (Integer)	Earliest allowable start time by previous agent
	next (String)	The agent that processes the next task
	prev (String)	The agent that processes the previous task

3.2.3 Communication Between Java and JADE Agents

The sequence diagram depicted in Figure 24 shows how a CSE, Joe, would send to and receive a message from another CSE, Dan.

As shown in the diagram, the `SchedulingBehaviour` main thread of Joe's Java scheduling agent would write twice across a socket to the JADE interface agent: once to indicate the recipient of its message (the agent called "Dan") and a second time with the String representation of the change, result, or other message type. "Dan" is placed in the ACLMessage recipient field and the message string in the content field. This ACLMessage is sent to the JADE interface agent named "Dan" using the HTTP message transfer protocol.

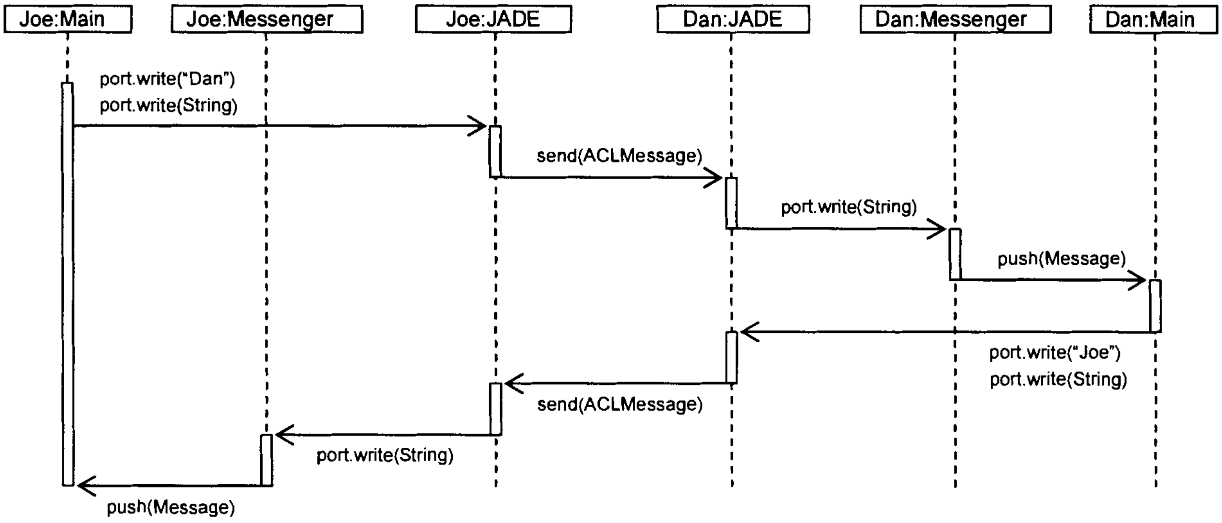


FIGURE 24. MESSAGE SEQUENCE DIAGRAM

Agent Dan will receive the ACLMessage and parse out the content field. It then connects as a client to the Java scheduling agent and passes this String across the open socket. Dan's `MessagingBehaviour` thread reads the incoming socket, converts the String to a Message type, verifies its `rcv` number, and pushes it to the end of the internal message queue. The main scheduling thread of Dan's Java scheduling agent will eventually pop this message off the message queue, process it, and return a reply by connecting directly to its JADE interface agent and writing "Joe" followed by the reply message String to the opened port.

Again converted to an ACLMessage, information is passed to Joe's JADE interface agent, then as a String to its `MessengingBehaviour`, and finally as a message to its main processing thread where it will be processed once popped off the internal message queue.

In this chapter, I have examined the implementation of the sliding window scheduling system using Java for JADE agent systems. Many of the details of the algorithm's design, including the synchronization of agents, were handled with the implementation in mind and an understanding of the additional considerations that implementation in JADE requires (for example the latency of messages between agents). Because of its modular design, the Java scheduling agent can be used in conjunction with an interface module for any multi-agent platform (JADE, JXTA, JINI, FIPA-OS, etc.).

4 SIMULATION RESULTS

In this chapter, I will examine the simulation results of agent systems using the proposed scheduling algorithm. The impact of hardware components will be investigated, as well as the effects of system growth, both in terms of the number of available resources and the number of requested jobs. Most importantly, I will present the results obtained when the sliding window approach is utilized.

4.1 Resource Utilization

In scheduling problems, there are a number of performance measures that can be examined and optimized. Some schedulers attempt to minimize lateness (how much time elapses between the order's due date at its actual delivery) [29]. Others are concerned with minimizing earliness (how much time elapses between the order's delivery time and its actual due date, the opposite of lateness). Still others minimize both earliness and lateness simultaneously, often using penalty functions [30][31][32][33][34]. For some complex systems where the resources themselves are variables in the equation, it is important to determine the fewest resources required to deliver all orders on time.

In the problem examined here, the resources and the tasks delegated to them are assumed to be fixed elements of the process and the performance measure that we look to maximize is the *utilization* of our resources, defined as:

$$\text{Utilization} = \frac{\text{Time resources are in use}}{\text{Total time required to complete all orders}} \times 100\% \qquad (3)$$

Consider the simple system of two agents shown below (Figure 25).

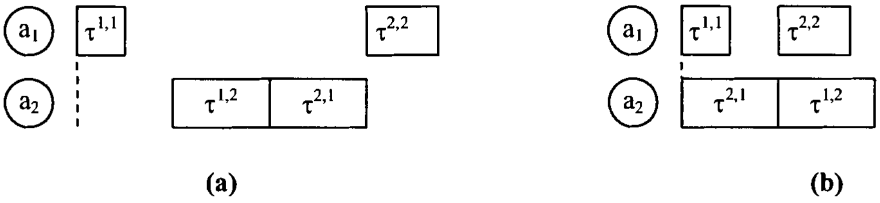


FIGURE 25. SCHEDULES FOR A SIMPLE TWO-AGENT SYSTEM

Assume tasks $\tau^{1,1}$ and $\tau^{2,2}$ each have a duration of 30 minutes, while tasks $\tau^{1,2}$ and $\tau^{2,1}$ each have a duration of 20 minutes. Both schedules are equally feasible (i.e. neither violates the precedence or resource constraints). Yet the schedule depicted in b) clearly completes all orders in less time. Comparing utilizations yields the following:

TABLE 3. UTILIZATION CALCULATIONS FOR SIMPLE TWO-AGENT SYSTEM

SCHEDULE	WORKING TIME	TOTAL TIME	UTILIZATION
a	100 minutes	200 minutes	50%
b	100 minutes	120 minutes	83%

An important observation to make regarding the preceding example is that 100% utilization, while desirable, may not be achievable for all scheduling configurations. Given the precedence and task durations in the problem above, there is no better way to order our tasks than Schedule B, which only yields 83% utilization. The challenge with each scheduling problem examined here is to determine the ordering of tasks that gives the best utilization and for each agent to do so with the limited knowledge it has of the overall system.

4.2 Task Selection Criteria

As mentioned in Section 2.2.1, there are often multiple operations eligible to be moved into scheduling gaps during the coarse sorting phase. When this situation arises, a number of selection criteria can be used to choose a candidate task to fill the gap, based on the limited information agents possess. Three criteria tested are:

- First-come first-served, which shows preference based on order number
- Furthest move, which moves the task whose start time will be most improved
- Best fit, which selects the task whose duration is closest to the gap width

Each of these criteria can be evaluated based on information that the agent already possesses.

To determine which of these criteria is best for our purposes, a random schedule of twenty-five agents, each with twenty-five tasks, was tested twenty-five times using the simulation application that will be discussed later in this chapter. The following scatter plot (Figure 26) and table (Table 4) demonstrate the performance of these three criteria. The vertical axis of the plot shows the schedule utilization and the horizontal axis shows the total number of sorting iterations required.

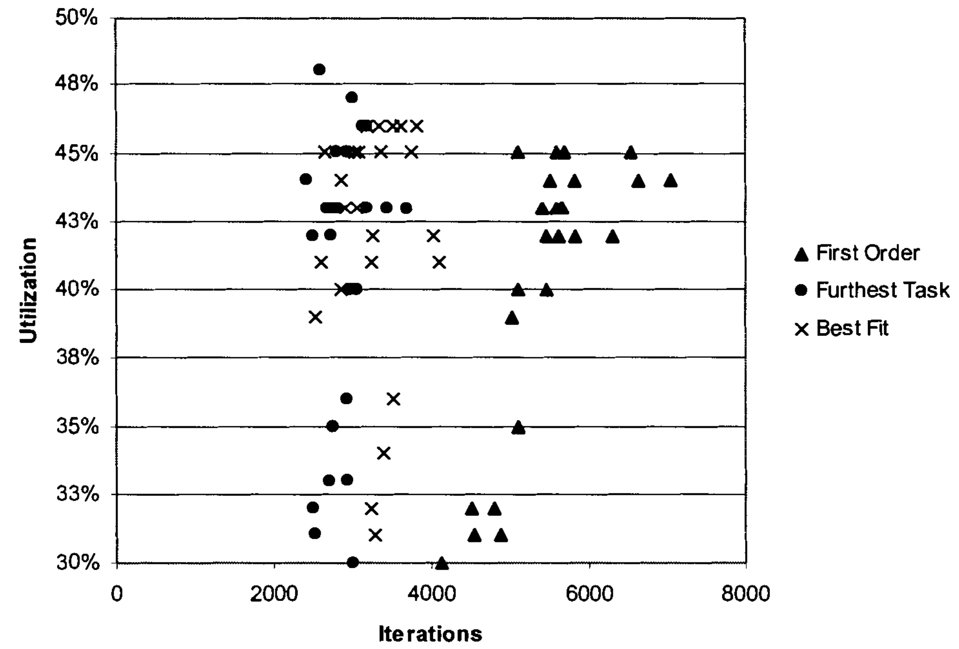


Figure 26. Coarse Sorting Selection Criteria Performance

Table 4. Coarse Sorting Performance Summary

Selection Criterion	Mean Trials	Standard Deviation	Mean Utilization	Standard Deviation
First Order	5488	680	40.2%	5.1%
Furthest Task	2902	304	40.5%	5.4%
Best Fit	3299	438	41.8%	4.4%

Not surprisingly, the performance of all three sorting criteria in terms of utilization is approximately equal, since the optimal arrangement of tasks is a function of the task distribution and not of the heuristic method used to approach the optimum. However, the furthest task selection criterion required noticeably fewer iterations to improve utilization than the other two. Based on the number of iterations, the second of these selection criteria (selecting the latest task capable of filling a scheduling gap) is used for the remaining simulations presented here.

4.3 Hardware Impacts on System Performance

The implemented system allows agents to negotiate an optimized schedule quickly and effectively. While one may expect total execution time to be an appropriate measure of how quickly the algorithm converges, different systems and networks of agents will yield different execution times based on the speed of the processors involved and their connections to the network.

To illustrate the impact of hardware choices on system performance, four agents running on four separate workstations were given a varying number of tasks to schedule. For each test case, the system was simulated ten times and no sliding window was utilized. The same test cases were then simulated when one of the computers was replaced with a slower machine. The resource configurations are described below in Table 5.

TABLE 5. HARDWARE TEST PLATFORM DESCRIPTION

RESOURCE	CPU	CACHE	BUS SPEED	RAM
1	Intel P4 1.4 GHz	256K Advanced Transfer L2	400 MHz	256 MB
2	Intel P4 1.5 GHz	256K Advanced Transfer L2	400 MHz	256 MB
3	Intel P2 350 MHz	512K L2	100 MHz	256 MB
4	Intel P4 2.4 GHz	512K Advanced Transfer L2	400 MHz	512 MB
5	Intel P4 2.4 GHz	512K Advanced Transfer L2	400 MHz	512 MB

Test platform 1 consisted of computers 1, 2, 4, and 5. Test platform 2 consisted of computers 2, 3, 4, and 5. Because the total simulation time is dependent on the number of iterations required by the algorithm and the number of messages processed by each agent, the averages of all three measurements are tabulated below in Table 6 for the two platforms tested.

TABLE 6. HARDWARE SIMULATION RESULTS

	TEST PLATFORM 1			TEST PLATFORM 2		
JOBS	TIME (SECONDS)	ITERATIONS	MESSAGES	TIME (SECONDS)	ITERATIONS	MESSAGES
5	1.5	21	278	2.4	21	277
10	5.7	86	931	7.9	90	963
20	24.5	272	3680	25.8	260	3504
30	61.8	466	11157	61.7	439	10861
40	121.2	733	22702	142.4	784	24278
50	207.3	1074	43049	235.9	1015	42138

As Table 6 illustrates, the time required to simulate the system with the slower workstation increased on the whole. This increase was nominally larger for larger systems. However, since the algorithm is not deterministic and the time required to simulate is dependent both on the number of iterations required by the algorithm and the number of messages passed between agents, the relative increase of all metrics must be analysed.

Table 7 shows the relative increase in time, iterations, and messages between Test Platform 1 and Test Platform 2.

TABLE 7. HARDWARE SIMULATION RESULTS ANALYSIS

JOBS	TIME	ITERATIONS	MESSAGES
5	60%	0%	0%
10	39%	4%	3%
20	5%	-5%	-5%
30	0%	-6%	-3%
40	17%	7%	7%
50	14%	-6%	-2%

These results clearly show that the increase in time required to simulate the ten runs of Test Platform 2 exceeds both the increase in iterations and the increase in messages. We can therefore conclude that the increase in time is not simply a result of the simulation runs requiring a varied number of iterations and messages to complete, but must instead be a result of the change in workstations used.

Because of the dependence of simulation time on hardware configuration, the number of algorithm iterations (how many different orderings of tasks were evaluated by each agent) and the number of messages processed will provide a clearer indication of system performance. Therefore, the remaining figures and tables presented will focus on these metrics.

4.4 Initial Sliding Window Test Results

To evaluate the sliding window algorithm, a system of five agents was configured to schedule five, twenty-five, and fifty jobs with varying window widths. Each task was assigned a random duration between one and twenty-five time units. The window width parameter, Γ, was initially assigned a value of twice the mean task processing time, and doubled after every ten simulations up to a width of thirty-two times the mean processing time. Future references to window width in figures and tables will refer to the ratio of width to the mean task processing time.

4.4.1 Jade Test Results

With a single agent running on each of five networked computers, the iterations and messages required by the system to schedule all tasks were recorded. Figures 27 and 28 show the changes in iterations and messages required as a function of window width.

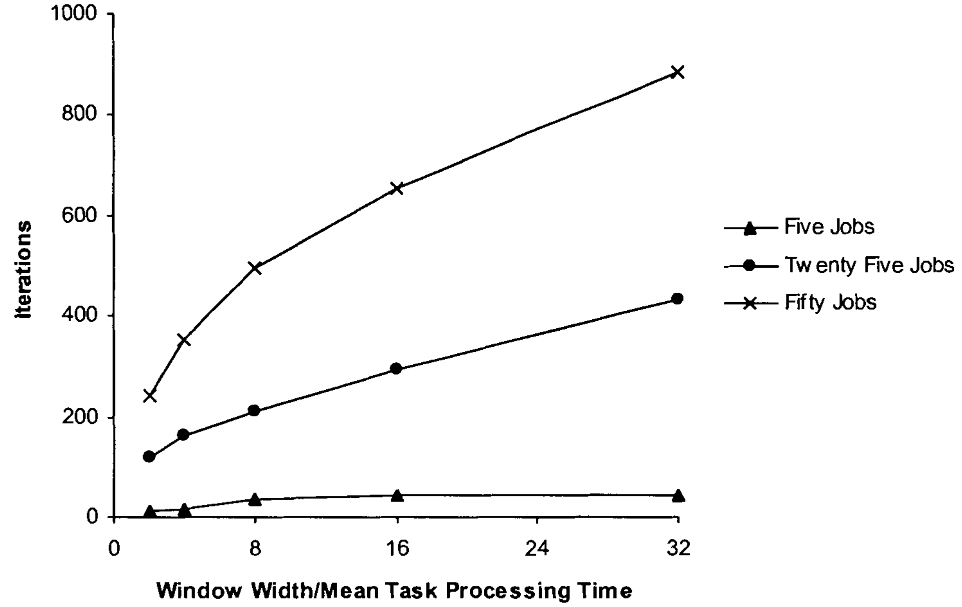


FIGURE 27. ITERATIONS REQUIRED BY JADE SIMULATIONS

As expected, though a large number of iterations and messages are needed for large window sizes, the algorithm shows a great savings in both iterations and messaging as the width decreases. This trend results from smaller windows allowing fewer task exchanges than large ones and leading to fewer propagated messages. Table 8 below illustrates the impact of the sliding window on the standard deviation of iterations, where the width values denote the ratio of window width to mean task duration. With fewer eligible tasks to exchange, smaller windows lead to more consistent decision-making among agents.

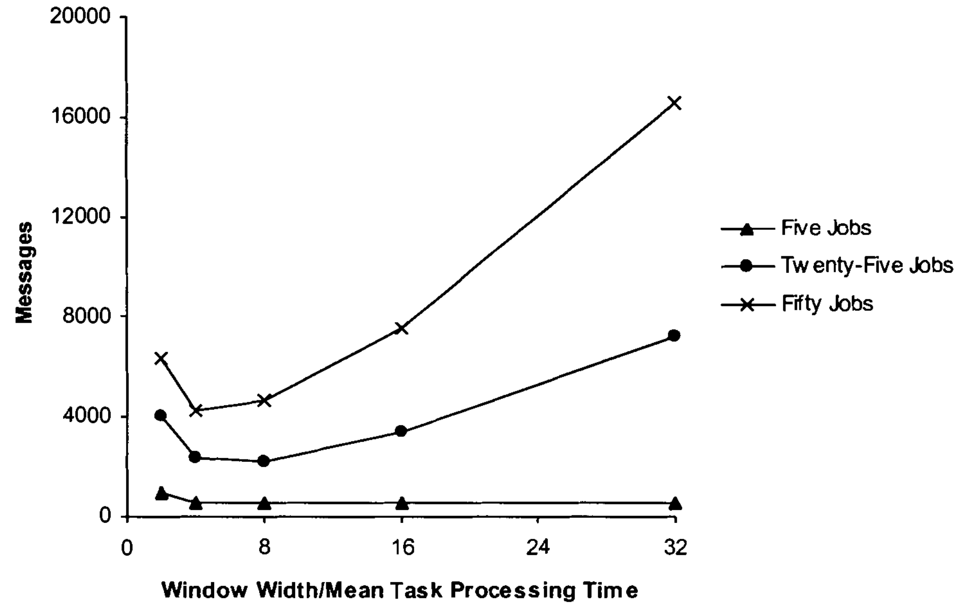


FIGURE 28. MESSAGES REQUIRED BY JADE SIMULATIONS

TABLE 8. STANDARD DEVIATION OF SIMULATIONS

		JOBS		
		5	25	50
WIDTH	2	0.0	2.1	11.5
	4	0.0	13.2	23.8
	8	0.6	13.8	18.7
	16	2.3	30.5	38.1
	32	2.2	105.2	78.5

By reducing the computational load, simpler and less expensive processors will handle the scheduling of tasks for resources using a sliding window approach. Concurrently, the sliding window technique can reduce the message handling requirements of our agents and thereby limit the amount of message traffic that consumes our network. Not only can more agents connect to a limited-capacity network, agents with slower connections to the network will be less heavily burdened by incoming messages.

4.4.2 Distributed Network Simulation Results

Because of the limited number of workstations available in the IDEA lab for testing and the desire to easily evaluate changes to the algorithm during development, a Windows application was written in C++ to emulate the performance of a distributed network of JADE agents running the sliding window scheduling algorithm. This application was designed using Borland C++ Builder to provide a graphical user interface for display of agent messages and the current system schedule in Gantt chart form. The simulator executes each agent on its own dedicated thread and all messages passed between agents are consistent with the messages used in the JADE implementation. As a result, it can be used to simulate systems with many more agents and many more jobs than are currently possible in the IDEA laboratory environment.

To verify that the simulator could be used to test the sliding window algorithm on larger systems, the same simulations performed in Section 4.4.1 were again performed using the Windows simulation application. The results of these simulations are superimposed on Figures 29 and 30 in the figures below. The dashed lines represent the results of distributed JADE simulations presented earlier. The solid lines show the results of ten simulation runs for each window width.

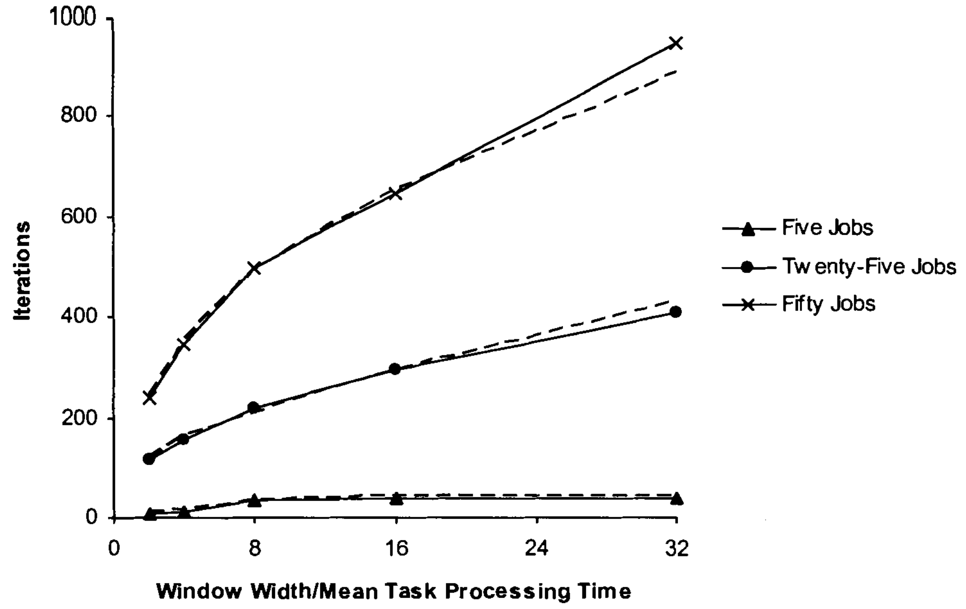


FIGURE 29. ITERATIONS REQUIRED BY SIMULATOR APPLICATION

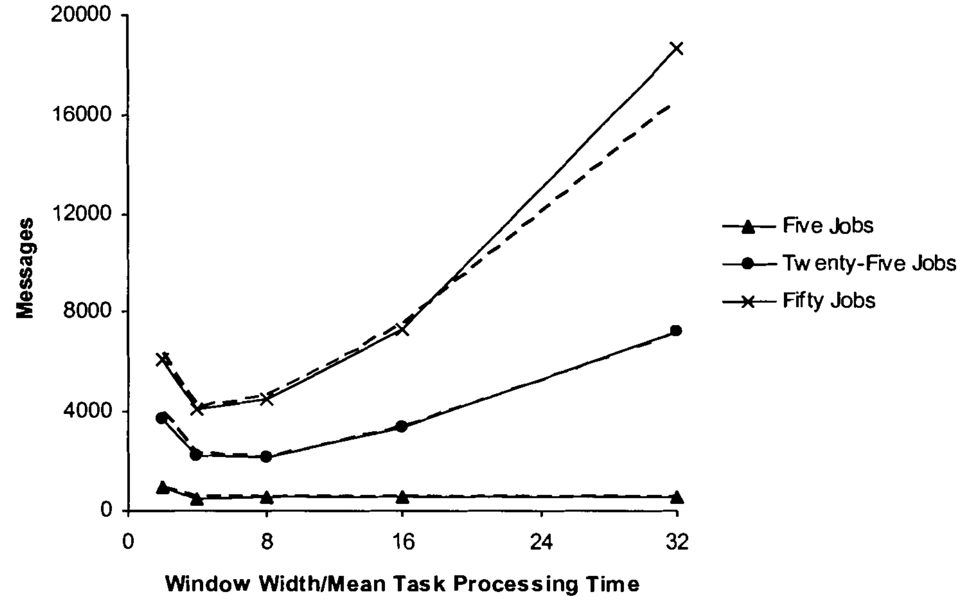


FIGURE 30. MESSAGES REQUIRED BY SIMULATOR APPLICATION

As can be seen, the simulator application closely replicates the results of the JADE simulations. The discrepancy that exists for fifty jobs using the largest window tested is due to the non-deterministic nature of the algorithm itself. For larger systems and larger windows, many decision paths will lead to the same optimal solution schedule. For the ten JADE simulations performed with the largest system, the standard deviations in iterations and messages were roughly 80 and 1700, respectively. Given that the results of the Windows simulations lie well within a single standard deviation, the simulator application can be considered a sufficient modeller for testing the sliding window algorithm on larger systems.

4.5 Impacts of System Dimensions on System Performance

As the size of systems increases, the computational and messaging loads increase significantly despite the use of distributed methods to reduce these stresses. The two most significant variables subject to change in the systems examined here are the number of resources that coordinate together and the number of tasks that they attempt to coordinate. Changes to each of these variables impact the scheduling algorithm and the messaging requirements differently. In this section, we isolate these two factors to show their contributions to the processing and message handling requirements of scheduling agents.

Using the simulator application, tests were performed to give an indication of how the number of resources in a distributed system affects the computational and communication bandwidth requirements. Using the basic coarse and fine sorting scheduling technique described earlier (without the sliding window approach), the average number of messages and task sorting iterations were recorded for twenty-five random systems with increasing numbers of resources. Each simulated system has ten jobs with each resource performing one task for each job. So the total number of tasks in these simulations is growing linearly with the number of resources, but the workload of each individual agent in the system remains constant: ten random tasks. Increasing the number of resources in this way is analogous to adding extra stages in the construction of each part. The figure below presents the averages of the data collected.

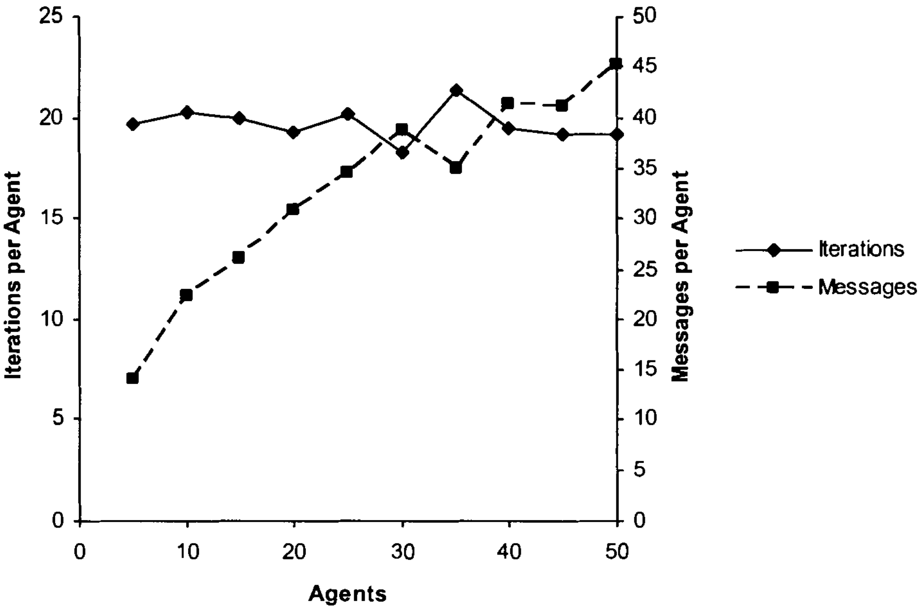


FIGURE 31. COMPUTING AND COMMUNICATION FOR INCREASING AGENTS

As the figure indicates, the number of trial solutions evaluated per agent does not significantly increase as the population of agents grows. Since each agent has the same number of tasks to evaluate, they will have the same number of potential exchanges regardless of the number of peers on the system. The significant increase is in the messages passed. With additional tasks for each job, each change requires additional propagation to be made. Furthermore, more system agents lead to more peer relationships and therefore additional synchronization messages for each resource.

Simulations were also performed to monitor the impact of job numbers to system performance. In these systems, there were twenty-five agents that each performed a single task for each order. The number of orders was allowed to vary between five and fifty to give an indication of the additional computations and communication demanded by the number of jobs.

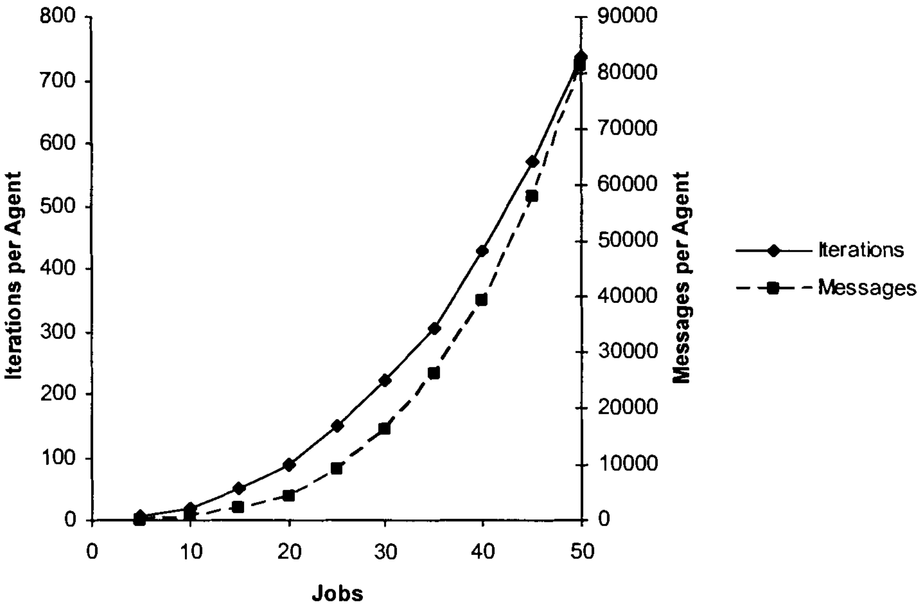


FIGURE 32. NUMBER OF ITERATIONS AND MESSAGES FOR INCREASING ORDERS

As expected, since the number of jobs for each agent dictates the number of possible orders to be tested, the number of iterations of the scheduling algorithm grows. For longer sequences of tasks, the number of messages required to propagate changes increases. These increases are exponential.

Growing systems clearly require a large number of trial solutions on the part of each agent and a large number of inter-agent messages to achieve an optimized schedule. For systems of hundreds of agents and thousands of jobs, these results clearly show the need for the sliding window alternative.

4.6 Impacts of Window Width on System Performance

Using the simulator application, curves similar to those presented in Section 4.4 were produced for a larger ten-agent system. Again jobs were randomly distributed between agents and each window width was tested ten times for each of the five configurations. These curves are shown in Figures 33 and 34.

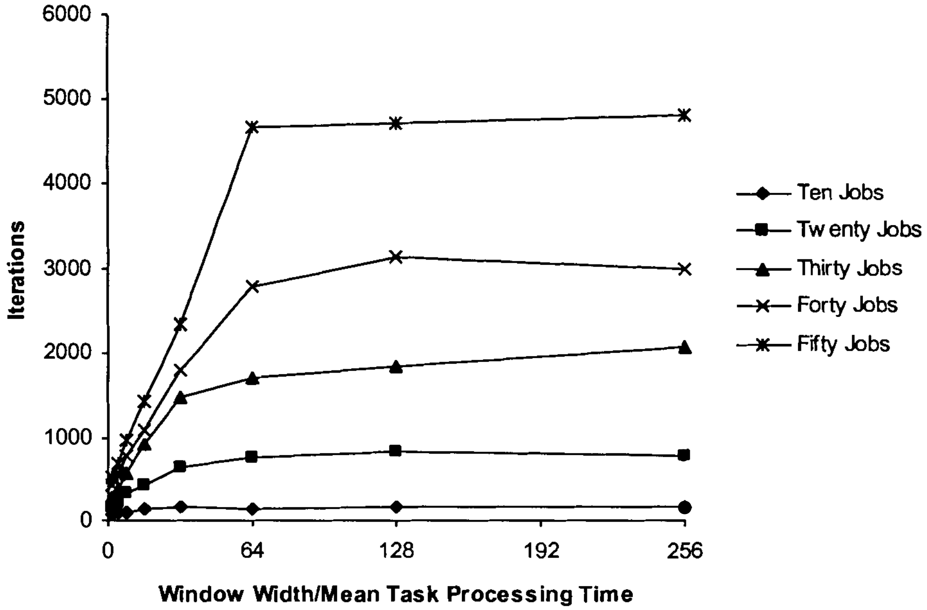


FIGURE 33. ITERATIONS REQUIRED FOR TEN-AGENT SYSTEM

These curves indicate that the total number of iterations and messages required by the algorithm begins to level off as the window size gets large. Beyond the value of Γ that allows the optimized schedule to fit into a single window, there is little change in performance. For relatively small window sizes, the number of iterations and messages decreases substantially. However, a closer look at the data (Figure 35) reveals that for small windows sizes, below eight times the mean task duration, the total messages required grows. This effect is due to the large number of synchronizations required.

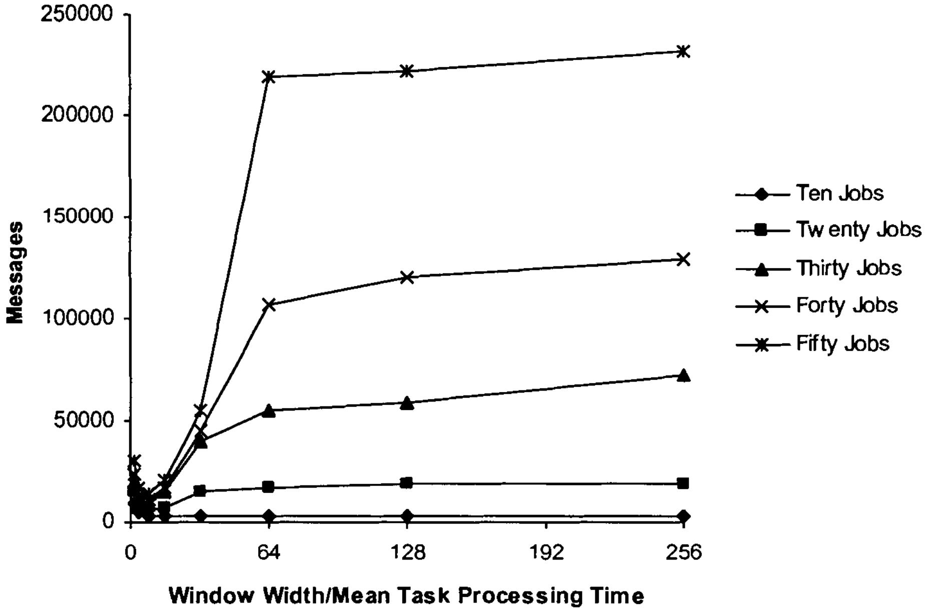


FIGURE 34. MESSAGES REQUIRED FOR TEN-AGENT SYSTEM

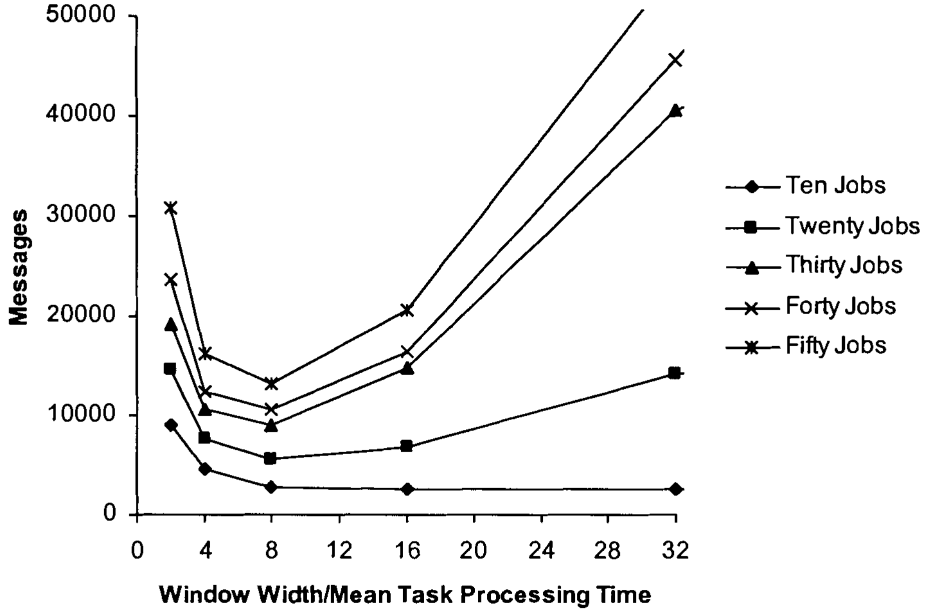


FIGURE 35. COMPARISON OF MINIMA OF MESSAGES CURVES

But the improvements in messaging and processing provided by sliding window usage is only beneficial if it leads to schedules with satisfactory utilization. Figure 36 shows the utilization attained in the simulations of the ten-agent system.

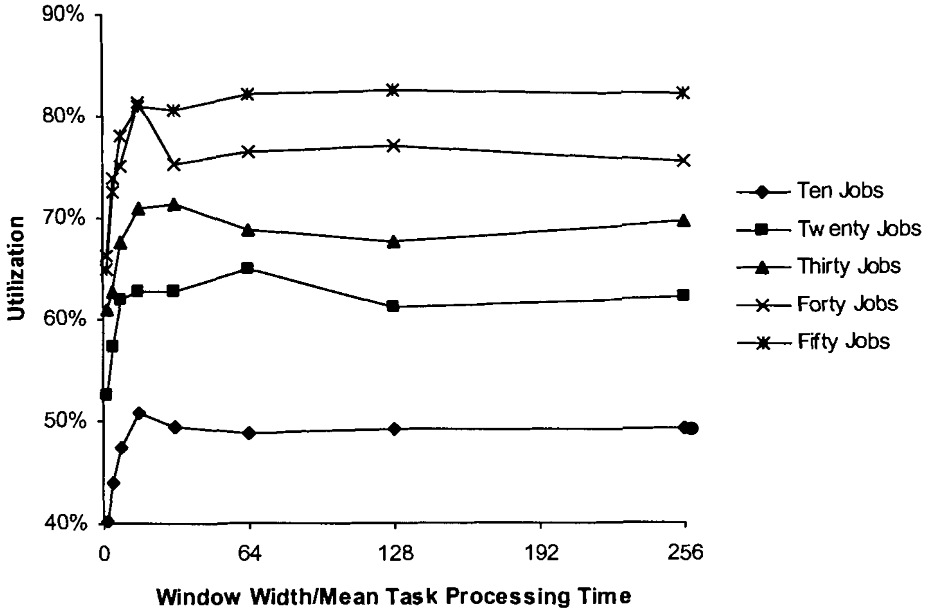


FIGURE 36. UTILIZATION ACHIEVED BY TEN-AGENT SYSTEM

The poorer utilization achieved using small window widths is expected. As the window size becomes small – near or less than the maximum width of a single task – the coarse sorting algorithm may no longer be able to select the best of several candidate tasks to fill a particular gap, but instead may be forced to choose the only one eligible. Similarly, small windows also limit the number of fine sorting possibilities. The overall result is a scheduling process that has limited potential to make great improvements through sorting and, therefore, will produce a schedule with poorer utilization. As the window size grows, the solution approaches infinite window scheduling, a scheduling of all tasks with no limited window of focus, and utilization levels off.

The system was then simulated using full optimization (i.e. coarse and fine sorting with no applied sliding window). The average performance of the system for each distribution of tasks is shown in Table 9.

TABLE 9. INFINITE WINDOW SCHEDULING RESULTS

JOBS	ITERATIONS	MESSAGES	UTILIZATION
10	179	3973	49%
20	711	20679	63%
30	1895	91344	70%
40	3256	189375	80%
50	5206	353864	82%

As the sliding window width grows, the iterations required by the algorithm and the utilization factors achieved approach the numbers obtained through full optimization. However there are significantly fewer messages needed by large windows. Because the sliding window technique allows reordering with a partly infeasible schedule, it bypasses the near many thousand extra messages that initial scheduling requires. With window widths of eight times the mean task duration, this difference in operation translates into message savings of up to 96%.

A promising observation is that the utilization of our system quickly reaches its final value for all five curves depicted above, achieving the best possible optimization even using relatively small windows. This result is very encouraging: a great improvement in both computational and messaging requirements with little degradation in resource utilization.

4.6.1 Determining an Ideal Window Width for Scheduling Systems

Being able to predict the best choice for window width for a given system would be beneficial. The first step in achieving this goal is to understand the reason for the minimum in our message curves. Using the utilization data above, it is possible to compute the impact of synchronization messages on the overall message passing of our systems. Considering that the mean task processing time is μ, an approximation of the total processing time for N tasks per agent is $N\mu$. We could then approximate the total makespan of our tasks as

$$\text{makespan} = \frac{N\mu}{u(n)} \tag{4}$$

where $u(n)$ is the utilization of the system achieve using a window width ratio of n. The number of windows of width $n\mu$ required to complete sliding window scheduling would be computed as

$$\text{windows} = \text{ceiling}\left(\frac{N\mu}{n\mu u(n)}\right) = \text{ceiling}\left(\frac{N}{nu(n)}\right) \tag{5}$$

The ceiling function returns the next integer greater than the argument provided. Since there are eight synchronization messages required between each peer for each window, the total number of synchronization messages required is then

$$\text{synchronization messages} = 8P\left[\text{ceiling}\left(\frac{N}{nu(n)}\right)\right] \tag{6}$$

where P is the total number of peer relationships between agents given the current task distribution. Using equation (6) and the peer, message, and utilization data for our simulations, the number of synchronization messages required by our systems was calculated for each window width. The relatively contribution of synchronization messages to the total messages processed by agents was also calculated and was then tabulated in Table 10.

TABLE 10. RELATIVE CONTRIBUTION OF SYNCHRONIZATION MESSAGES

		TASKS N				
		10	20	30	40	50
WIDTH n	2	97%	99%	94%	94%	91%
	4	88%	85%	81%	81%	80%
	8	74%	64%	48%	47%	49%
	16	51%	21%	15%	18%	14%
	32	26%	5%	4%	3%	3%
	64	25%	4%	1%	1%	<1%
	128	22%	4%	1%	1%	<1%
	256	22%	4%	1%	1%	<1%

For small window widths relative to the mean task length ($n = 2, 4$), synchronization messages account for the vast majority of messages processed by agents (up to 99%). As the window width grows, their contribution becomes less and less. Eventually, for large agent systems, they can account for less than one percent of the total number of messages received. All other messages processed are change and result messages, used to schedule and sort the tasks themselves.

The point at which the number of synchronization messages is roughly equal to the number of scheduling messages aligns with the minima of Figure 35 for each of the five curves provided. For the ten-agent systems with fewer tasks ($N = 10, 20$), the minima occur between 12 and 16 times the mean task duration. For systems with greater tasks ($N = 30, 40, 50$), they occur at roughly 8 times the mean task duration.

To calculate the ideal window width, we would need to be able to model the number of messages required for scheduling and sorting as a function of the window width ratio.

Given that N and P are properties we can measure before scheduling the system, based purely on the distribution of tasks, we should ascertain whether utilization and total messages follow some pattern for systems of a chosen dimension. Though a thorough analysis of systems of varying dimension is beyond the scope of this work, determining whether the analysis above holds, for just the sample systems presented or for any similar systems of the same dimension, is an important step.

A final simulation of the ten-agent system was performed. Each of the window widths was simulated twenty-five times with a different set of thirty jobs distributed randomly between the ten agents. Figures 37 and 38 below illustrate the range of results achieved by random systems. The maximum and minimum messaging and utilization curves are provided, in addition to the average performance of ten-agent thirty-job systems.

From these figures, it is clear that there is a vast range of system performances using the sliding window algorithm. Some systems required 50% fewer messages than others using the same window size. Utilizations varied by as much as 17%.

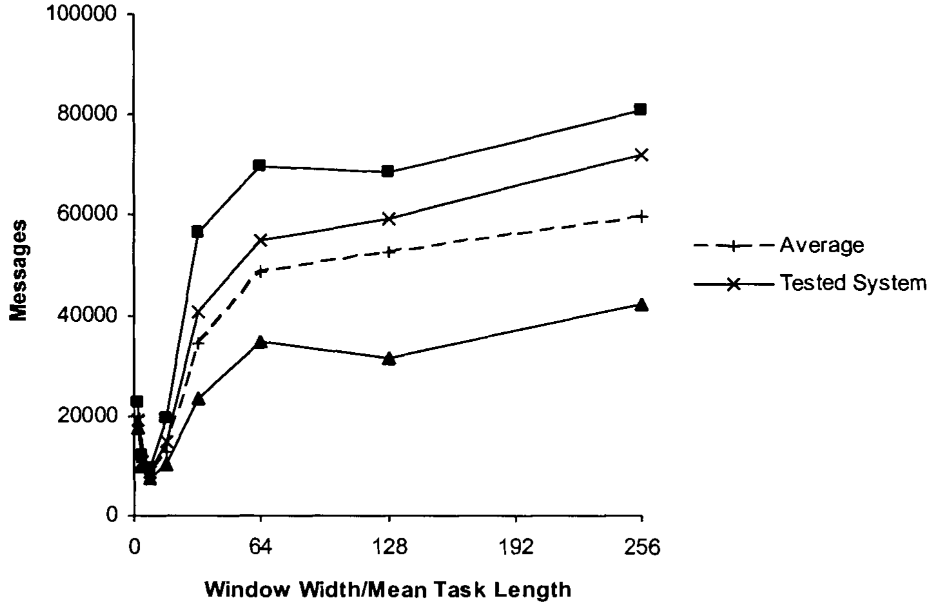


FIGURE 37. MESSAGE REQUIREMENTS FOR TEN-AGENT THIRTY JOB SYSTEMS

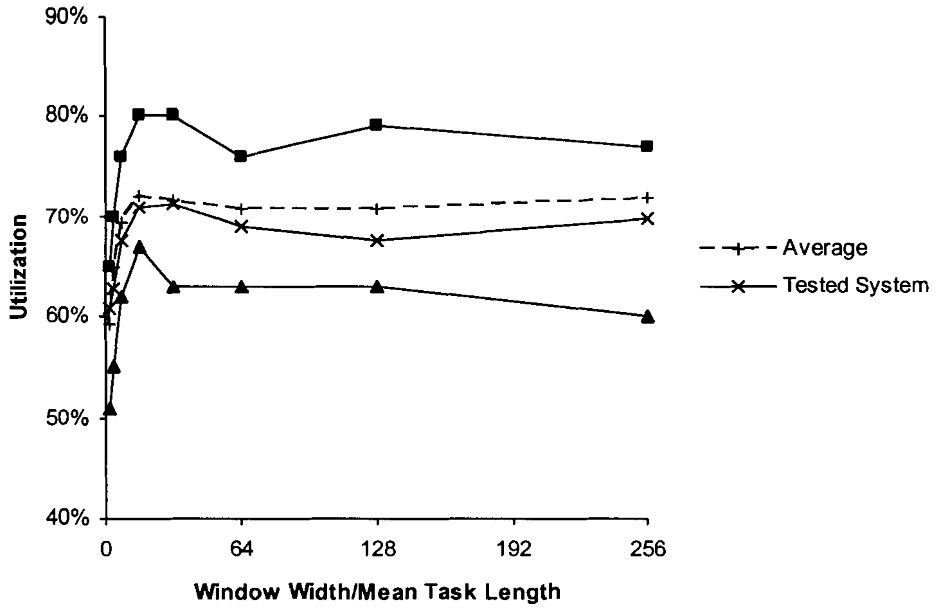


FIGURE 38. MESSAGE REQUIREMENTS FOR TEN-AGENT THIRTY JOB SYSTEMS

There were consistently between 88 and 90 peer relationships between agents in these random systems, although it is possible for full connectivity between all agents with as few as eighteen peer relationships. The downside of these results is that the messages and utilizations achieved do not appear to be predictable for systems of consistent dimension. Without this information, calculating the optimal window size becomes much more difficult. Yet, in spite of the variance in both messages and utilization, systems of ten agents with thirty jobs each experience a messaging minimum at roughly the same point. As a result, the potential still exists for an ideal window width to be determined based purely on the system configuration.

In this chapter, I have measured the performance of the sliding window scheduling algorithm and shown that while it provides great benefits in terms of computational requirements and communication bandwidth savings, these benefits do not come at the expense of poorer resource utilization. As systems become large and new orders are added, the significance of these benefits becomes greater still.

5 CONCLUSIONS

5.1 Future Work

With a project such as this one, there are many directions in which future effort may be taken to improve the performance, robustness, and relevance of this work, and many areas of research that would benefit from the research already performed to this point. In this section, we will briefly acknowledge some of these potentials.

5.1.1 Improvements to Performance

One drawback to the sliding window approach to scheduling is that all information about the length of the series of tasks beyond the window's end is discarded (i.e., a job with only one task beyond the window is treated the same as a job with one hundred subsequent tasks). Consequently, the result message is no longer an accurate reflection of the impact of each change on the system. As scheduling concludes, the potential exists for a long string of tasks to find itself at the end. This condition greatly reduces the utilization of the overall system, since agents will be left idle while the single tasks of the final job are performed. A feature that allows the agents to look ahead to subsequent tasks could improve performance.

Meanwhile, as noted in Chapter 4, there appears to be an optimal window width corresponding to systems of agents with a given task distribution and peer interaction.

Finding the relationship between peer relationships, task distributions, and the window width that will best optimize the schedule is an important area for future research.

5.1.2 Improved Robustness

One of the major requirements of multi-agents systems is their ability to adapt to change. Though the system presented here would be responsive to new orders, the one condition to which it is not yet responsive is agent failure.

As mentioned, a major advantage to the recursive propagation technique of message passing for scheduling is that it eliminates the need for a central blackboard and makes no single agent more critical to system performance than any other. But the implementation of the system to utilize this advantage has not yet been realized. When an agent fails, suddenly the chain of tasks that constitutes a given order is broken. These lost tasks must be reassigned to available resources to allow the system to continue scheduling.

Modification of the existing system to handle these events is certainly possible, though it would require an additional fault-response behaviour of the agents. Tasks could contain not just the names of the agents who perform adjacent operations, but also the details of those operations so that, should an agent fail, the peers of the failed agents would be able to reassign the unclaimed tasks to other resources. The JADE yellow page functionality should play a vital role in this, allowing agents to seek out others on the system with the necessary machine functions to complete the given task and assign it accordingly. By receiving information from both the agent that followed and the agent that preceded the now unclaimed task, the new resource will be able to reconstruct all task information and assign it an appropriate start time so that scheduling can continue, virtually unaffected.

5.1.3 Broadened Relevance

As described in the opening chapter, the problem addressed in this thesis is that of a linear progression of tasks as shown in Figure 39.

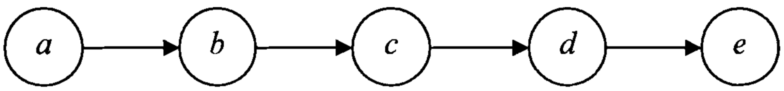


FIGURE 39. LINEAR TASK SEQUENCE FOR A SINGLE JOB

In fact, in scheduling resources (whether in manufacturing, human resources, transportation, or a number of other industries), there is a more complicated dependence of tasks for a single job. Potentially, a single resource may have to await the completion of several agents before its task can begin and, likewise, may have a number of other tasks dependent on a single task of its own (Figure 40).

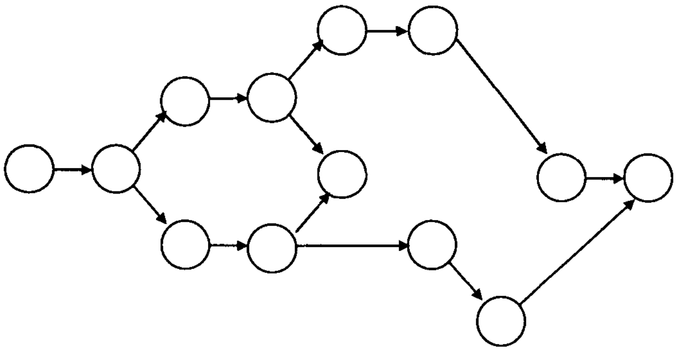


FIGURE 40. NON-LINEAR TASK SEQUENCE FOR A SINGLE JOB

These complex job configurations are currently unmanageable for the algorithm presented here. But by modifying the structure of a task so that each contains a *list* of dependent and subsequent agents and, additionally, a list of minStart values, the algorithm would be able to make decisions based on the agents that constitute the critical path for an order. Once this change has been implemented, the areas of application for this distributed scheduling approach will become far-reaching and the potential for this technique to address complex scheduling problems will greatly improve.

Furthermore, some real-world applications of scheduling require that idle time be built in to the schedules of resources. These periods of rest prevent excessive stress on machines and components. A future system that included rest periods would be valuable.

5.1.4 Complementary Research

As mentioned earlier, concurrent to this work on distributed scheduling, further investigation into distributed networks and problem solving is being performed by other students in the Intelligent and Distributed Enterprise Automation (IDEA) Laboratory at Simon Fraser University. In fact, the IDEA laboratory is, through the work of its students and research staff, working on all required levels of development for distributed wireless solutions (Figure 41).

At the device level, Gary Wong and Peter Bai are constructing the physical hardware and firmware to comply with 802.11x standards of wireless communication between devices. Edward Chen is conducting an investigation of the JADE agent framework and its effectiveness in multi-agent system implementation [28]. His research emphasizes the steps required in developing wireless distributed agent systems using JADE and compare the performance and overhead with other agent frameworks.

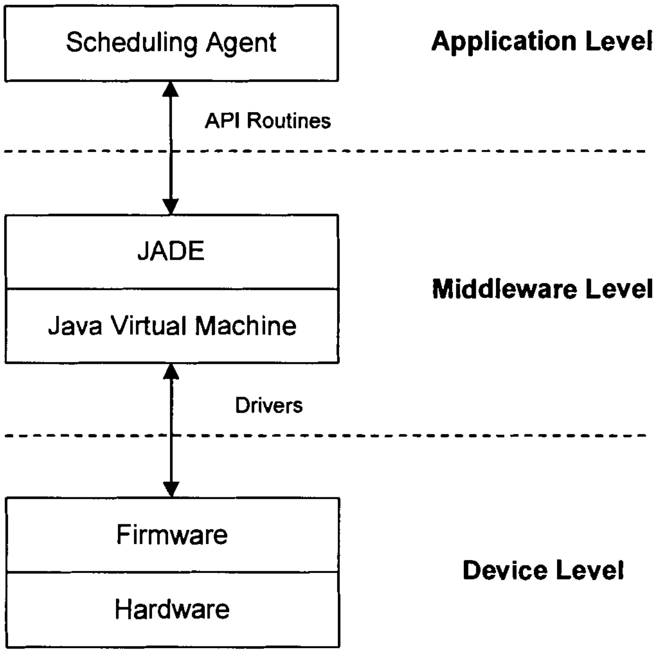


FIGURE 41. LEVELS OF AGENT SOFTWARE IMPLEMENTATION

Development of contract nets and distributed energy resources by another IDEA Laboratory student Zafeer Alibhai [35], as well as the distributed scheduling software detailed here, provide the application-level basis to properly demonstrate the usefulness of a wireless system. Eman Elghoneimy and Ozge Uncu are applying multi-agent system methodology to automate the operation of a rough mill in the manufacturing of wood frame windows for a Canadian company.

Lastly, Colin Ng and Steven Chen are developing VNET, a distributed network simulation platform to allow applications for wireless networks to be tested in a laboratory setting [27]. VNET will allow logical links between physically connected devices to have built-in intelligence and variance.

Despite being physically networked, distinct communication routes can be defined by VNET network administrators that dictate the only existing message pathways. These pathways through neighbouring nodes will model the communication infrastructure of existing peer-to-peer and wireless networks. By allowing links between nodes to be time-varying (periodically becoming unavailable and introducing occasional delays or bad message packets), many anomalies of wireless networks can be simulated.

Together, these projects could allow physically separated devices to utilize the message-passing protocol of JADE and other agent environments to demonstrate how a distributed scheduling system implementation could be used in a wireless network environment. Large networks of devices could be tested and real simulation data collected that would accurately model the real-world network environment in question. Furthermore, the VNET could thoroughly test the distributed sliding window scheduling algorithm and its response to both agent and communication link failure.

5.2 Summary

This thesis has demonstrated an improved technique for distributed scheduling that reduces the complexity of the traditional scheduling problem and correspondingly the communication overhead of typical agent-based approaches, without a significant degradation of resource utilization. The sliding window approach to scheduling presented here is also a dynamic approach, allowing agents to add new orders on the fly and handle the challenges of an ever-changing order list. In addition to providing the benefits of distributed processing and computation, the use of recursive propagation to update schedules also allows the communication itself to be decentralized, eliminating dangers of failure that accompany blackboard-related systems.

Using JADE, the algorithm outlined here has been demonstrated for agents on physically separated devices and, in future, can be used to test wireless networks of distributed agents. Together with the other research performed by students of the Intelligent and Distributed Enterprise Automation Laboratory at Simon Fraser University, composite scheduling entities implemented in Java for JADE agent systems will be important tools to illustrating the growing impact of decentralized problem solving approaches to many commercial applications.

6 REFERENCES

[1] K. Baker, *Introduction to Sequencing and Scheduling*, Durham, USA, 1974.

[2] H. Iima, R. Kudo, N. Sannomiya, and Y. Kobayashi, "An autonomous decentralized scheduling algorithm for a job shop process with a multi-function machine in parallel," *Proc. of the Third International Symposium on Autonomous Decentralized Systems*, Berlin, Germany, 1997.

[3] H. Iima, T. Hara, N. Ichimi, and N. Sannomiya, "Autonomous decentralized scheduling algorithm for a job shop scheduling problem with complicated constraints," *Proc. of the Fourth International Symposium on Autonomous Decentralized Systems*, Tokyo, Japan, 1999.

[4] Y. Foo and T. Takefuji, "Integer linear programming neural networks for job-shop scheduling," *Proc. of the IEEE International Conference on Neural Networks*, San Diego, USA, 1988.

[5] M. Ballicu, A. Giua, and C. Seatzu, "Job-shop scheduling models with set-up times," *Proc. of the IEEE International Conference on Systems, Man and Cybernetics*, Hammamet, Tunisia, 2002.

[6] G. Hasle, R. C. Haut, B. S. Johansen, and T. S. Ølberg, "Well activity scheduling – an application of constraint reasoning," *Proc. of Practical Applications of Constraint Technology*,London, UK, 1997.

[7] F. Hermann, K. Muller, and S. Engell, "FMS scheduling using branch-and-bound with heuristics," *Proc. of the 31st IEEE Conference on Decision and Control*, Tucson, USA, 1992.

[8] S. Fujita, M. Masukawa, and S. Tagashira, "A fast branch-and-bound algorithm with an improved lower bound for solving the multiprocessor scheduling problem," *Proc. of the 9th International Conference on Parallel and Distributed Systems*, Taiwan, 2002.

[9] H. Chen, J. Ihlow, and C. Lehmann, "A genetic algorithm for flexible job shop scheduling," *Proc. of the 1999 Conference on Robotics and Automation*, Detroit, USA, 1999.

[10] M. Watanabe, M. Furukawa, A. Mizoe, and T. Watanabe, "GA applications to physical distribution scheduling problems," *Proc. of the 26th Annual Conference of the IEEE Industrial Electronics Society*, Nagoya, Japan, 2000.

[11] K. Krishna, K. Ganeshan, and D.J. Ram, "Distributed simulated annealing algorithms for job shop scheduling", *IEEE Transactions on Systems, Man and Cybernetics*, Volume 25, Issue 7, 1995.

[12] Z-P. Lo and B. Bavarian, "Job scheduling on parallel machines using simulated annealing," *Proc. of the IEEE International Conference on Systems, Man, and Cybernetics*, Charlottesville, USA, 1991.

[13] H. Chen and P. Luh, "An alternative framework to Lagrangian relaxation approach for job shop scheduling," *European Journal of Operational Research*, 2003.

[14] R. Walker, "Introduction to the scheduling problem," *IEEE Design and Test of Computers*, Washington, USA, 995.

[15] J. Layden "The evolution of scheduling logic," *Plant, Canada's Industry Newspaper*, Toronto, Canada, 1999.

[16] D. Sabaz, W.A. Gruver, and M. H. Smith, "Distributed systems with agents and holons," *Proc. of the IEEE International Conference on Systems, Man, and Cybernetics*, The Hague, Netherlands, 2004.

[17] T. Vidal, B. Archimède, and T. Coudert, "Distributed forward checking for scheduling in flexible manufacturing cells," *Proc. of the 8th IEEE International Conference on Emerging Technologies and Factory Automation*, Juan les Pins, France, 2001.

[18] R. Hino, nd M. Toshimichi, "Decentralized scheduling in agent manufacturing system," *Proc. of the Second International Workshop on Intelligent Manufacturing Systems*, Leuven, Belgium, 1999.

[19] R. Hino, K. Izuhara, and M. Toshimichi. "Message exchange method for decentralized scheduling," *Proc. of the 4th IEEE International Symposium on Assembly and Task Planning*, Fukuoka, Japan, 2001.

[20] F. Bellifemine, G. Caire, A. Poggi, G. Rimassa, "JADE – A white paper," *EXP – In Search Of Innovation*, Volume 3, Number 3, Telecom Italia Labs, Turin, Italy, 2003.

[21] P. Vrba, "java-based agent platform evaluation," *Proc. of the 1st International Conference on Applications of Holonic and Multi-Agent Systems*, Prague, Czech Republic, 2003.

[22] R. Conway, W. L. Maxwell, and L. W. Miller, *Theory of Scheduling*, Mineola, USA, 1967.

[23] B. Jeremiah, A. Lalchandani, and L. Schrage, "Heuristic rules toward optimal scheduling," Research Report, Department of Industrial Engineering, Cornell University, USA, 1964.

[24] S. Logie, D. Sabaz, and W.A. Gruver, "Combinatorial sliding window scheduling for distributed systems," *Proc. of the IEEE International Conference on Systems, Man, and Cybernetics*, Washington, USA, 2003.

[25] S. Logie, D. Sabaz, and W.A. Gruver, "Sliding window distributed combinatorial scheduling using JADE," *Proc. of the IEEE International Conference on Systems, Man, and Cybernetics*, The Hague, Netherlands, 2004.

[26] H. Gou, B. Huang, W. Liu, S. Ren, and Y. Li, "An agent-based approach for workflow management," *Proc. of the IEEE International Conference on Systems, Man, and Cybernetics*, Nashville, USA, 2000.

[27] C. Ng, D. Sabaz, and W.A. Gruver, "Distributed algorithm simulator for wireless peer-to-peer networks," *Proc. of the IEEE International Conference on Systems, Man, and Cybernetics*, The Hague, Netherlands, 2004.

[28] E. Chen, D. Sabaz, and W.A. Gruver, "JADE and wireless distributed environments," *Proc. of the IEEE International Conference on Systems, Man, and Cybernetics*, The Hague, Netherlands, 2004.

[29] D. Huang, Y. Zhu, Y. Zhao, and W. Wang, "A heuristic algorithm for minimizing the range of lateness and make-span on non-identical multi-processors," *Proc. of the 3rd World Congress on Intelligent Control and Automation*, Hefei, China, 2000.

[30] S. Tanaka, T. Sasaki, and M. Araki, "A branch-and-bound algorithm for the single-machine weighted earliness-tardiness scheduling problem with job independent weights," *IEEE International Conference on Systems, Man and Cybernetics*, Washington, USA, 2003.

[31] Y. Genke, W. Zhiming, and C. Oguz, "A branch and bound approach for earliness and tardiness penalty problem with distinct due dates," *Proc. of the 4th World Congress on Intelligent Control and Automation*, Shanghai, China, 2002.

[32] W. Li and W. Mengguang, "A hybrid algorithm for earliness-tardiness scheduling problem with sequence dependent setup time," *Proc. of the 36th IEEE Conference on Decision and Control*, San Diego, USA, 1997.

[33] H. Tamaki, H. Murao, and S. Kitamura, "A heuristic-based hybrid solution for parallel machine scheduling problems with earliness and tardiness penalties," *Proc. of the IEEE Conference on Emerging Technologies and Factory Automation*, Lisbon, Portugal, 2003.

[34] L.B.Valencia and G. Rabadi, "A multiagents approach for the job shop scheduling problem with earliness and tardiness," *Proc. of the IEEE International Conference on Systems, Man and Cybernetics*, Washington, USA, 2003.

[35] Z. Alibhai, R. Lum, W.A. Gruver, A. Huster, and D. Kotak, "Coordination of distributed energy resources," *NAFIPS 2004 Conference*, Banff, Canada, 2004