A SCHEDULING ALGORITHM FOR A REAL-TIME MULTI-AGENT SYSTEM

BY

ETHAN HODYS

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE

REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

IN

COMPUTER SCIENCE

UNIVERSITY OF RHODE ISLAND

2000

MASTER OF SCIENCE THESIS

OF

ETHAN HODYS

APPROVED:

THESIS COMMITTEE

MAJOR PROFESSOR _____

_____

_____

_____

DEAN OF THE GRADUATE SCHOOL

UNIVERSITY OF RHODE ISLAND

2000

Abstract

In recent years, research of software agents has gained a tremendous amount of attention. To date, very little of the research has involved agents that possess real-time constraints or operate within real-time systems. A real-time multi-agent system (RTMAS) would have certain unique characteristics versus both traditional real-time systems and multi-agent systems. More specifically, the scheduling algorithm for such a system would be capable of exhibiting a more robust scheduling algorithm than previously implementations. This stems directly from an agent's ability to possess multiple execution times for a given task. The change in execution time of these methods is proportional to the "quality" of the result produced. In such a system, before a set of tasks is declared "non schedulable", the quality of the results of scheduled tasks can be reduced in an attempt to achieve total "schedulability" for all tasks seeking execution. The greatest system utility is gained by executing all requested tasks while returning the best possible result within the assigned time slot. An exact definition of "quality" in regards to this paper is offered further in the reading. We have developed such an algorithm based on a heuristic that determines which tasks are the least costly to reduce in order to minimize the reduction in quality. We have also developed a model for real-time agents as well as a model for real-time agent scheduling.

Acknowledgement

I would like to thank Colleen and my parents for providing me with the endless support and love that is necessary to be truly successful at an endeavor such as this. In addition, I would like to thank my major professor Dr. Lisa DiPippo for her advice and expertise.

My gratitude is also extended to Dr. Fay Wolfe and the members of the RealTime Research Group for their time, advice and input. To Lorraine and Marge, thank you for making my experience at URI an easier and more enjoyable one.

CONTENTS

TABLES

FIGURES

1.0 Introduction

With the advent of distributed computing and the commercial success of the Internet, the efficient design and application of distributed software has taken on a more elevated role in Computer Science research. The object-oriented design methodologies that have been so successful for non-distributed applications have recently caused problems when applied to the development of specific distributed applications [1]. In particular, the object-oriented model 1) lacks the ability to handle open environments effectively and 2) cannot be applied to heterogeneous systems without tremendous difficulty. The development of "Software Agents", loosely defined as autonomous intelligent software entities, is an extension of the object-oriented paradigm that imbues traditional "objects" with intelligence allowing for a greater level of decentralization; this is a desirable characteristic in a distributed environment. An Agent-oriented paradigm is more appropriate for a distributed environment for additional reasons. First, the autonomy of the agents allows for more efficient communication and processing among distributed resources. Second, the flexible and responsive nature of agents provides benefits for real-time applications; yet another characteristic that is highly beneficial in a significant percentage of distributed applications. These benefits will be explored in detail later. Third, agents are highly extensible.

Applications such as information access, information filtering, electronic commerce, workflow management, and intelligent manufacturing are becoming increasingly common and necessary. The industry demand for such applications far exceeds the current supply, and to some extent, exceeds the current state of technology. The major contributing factor to this lack of supply is the inherent difficulty, using

current technology, in creating applications that work well in “open” environments. An open environment is one in which the sources of information are autonomous, heterogeneous, and updated (added or removed) dynamically. All the applications previously mentioned work in open environments. In addition, these applications need mechanisms in order to advertise, find, fuse, use, present, manage, and update information. A significant number of these applications inherently place timing constraints upon all or most of these mechanisms. The characteristics of an open environment dictate that the associated mechanisms must be both extensible and flexible. Since, as stated above, extensibility and flexibility are two of the benefits provided by software agents, it is natural that many researchers now view agents as an integral part to creating such mechanisms. The true power of software agents lies in their ability to provide these mechanisms in unpredictable environments.

A designer of distributed systems may want to begin taking advantage of the power of software agents, but generally would like to avoid having to learn an entirely new developing environment. This would negate the expertise and experience they have achieved from designing systems within their current development environment. In addition, learning an entirely new design paradigm and the software package to implement it is a significant economic cost that all rational developers seek to avoid unless absolutely necessary. Therefore, there is a high demand for extensions to current distributed computing specifications and developing environments that allow for the use of software agents.

This research began with the desire to provide CORBA developers with the ability to utilize software agents. The first goal was to design an algorithm for scheduling

the execution of software agents in a real-time environment. This scheduling algorithm could then be incorporated into the Real-Time CORBA specification. Because of the special characteristics of agents versus CORBA objects, a standard real-time scheduling algorithm would be too limiting, specifically during times of high resource contention. We have designed an algorithm for scheduling agents that works within a real-time environment and takes full advantage of an agent's flexibility though a concept called "load reduction". Although this alone will not allow CORBA developers the ability to utilize software agents, it is an essential first step.

2.0Agents

2.1 Key Characteristics

While it is true that a basic definition for an "agent" is impossible to present due to the varying differences in opinion, an all-encompassing list of agent properties can be presented to promote discussion and analysis. Researchers may disagree as to how such characteristics should be implemented or exactly how an agent should perform under certain conditions, but the properties themselves provide a finite set of characteristics that a generic agent defines. The tables below can be found in the introduction of "Readings in Agents" [1].

Property	**Range of Values**
Lifespan	Transient to Long-lived
Level of cognition	Reactive to Deliberative
Construction	Declarative to Procedural
Mobility	Stationary to Itinerant
Adaptability	Fixed to Teachable to Autodidactic
Modeling	Of environment, themselves, or other agents

Table 1: Agent characteristics: Intrinsic

Property	**Range of Values**
Locality	Local to Remote
Social autonomy	Independent to controlled
Sociability	Autistic, Aware, Team Player
Friendliness	Cooperative to Competitive to Antagonistic
Interactions	• Style/Quality/Nature with agents/world/both • Semantic level: declarative or procedural communications • Logistics: direct or via facilitators

Table 2: Agent characteristics: Extrinsic

There are two more properties that are extremely important to the discussion of agents. The first is autonomy; the second is intelligence. By nature, intelligence is a very difficult concept to define, both in regards to living beings and software. While a universal definition of intelligence for agents is far away, the most basic requirement is that agents have some form of rationality. Rationality in this context depends upon [7]:

- The performance measure for success
- What the agent has perceived so far
- What the agent knows about the environment
- The actions the agent can perform

An agent can be considered rational if for all possible events, the agent acts consistently to maximize its expected utility using the sum of its knowledge about the environment and what has been perceived. All the agents we have developed possess rationality as defined here, but no further claims or requirements will be made in regards to their intelligence.

In regards to software, autonomy is defined as a process running as a separate thread. There exist many different classifications of autonomy. For example, an agent may be autonomous in regards to the client, but may not be autonomous in regards to another agent. In this thesis a universal approach is taken in regards to autonomy; that is, autonomy is used to mean absolute autonomy. Any agent implemented or discussed here does not share process space with any other software entity in the system.

3.0 Real-Time Computing

3.1 Real-Time Artificial Intelligence

Traditionally, artificial intelligence techniques have not been utilized in real-time environments due to their highly unpredictable performance. Generally this is a result of the types of problems AI research has been focused on solving; that is, very difficult problems that often involve searching as a component of the solution method. Complex algorithms that incorporate searching are unpredictable mainly because it is never analytically clear how much of the search space must be seen in order to compute an answer [2]. A major step forward in real-time AI research began with the concept of approximate processing and approximate algorithms. To date, real-time AI research has been interested two main types of approximate algorithms: *Anytime Algorithms* and *Multiple (Approximate) methods.*

An anytime algorithm is an iterative refinement algorithm where a "default" answer is first generated and then refined through multiple iterations. It is also true that the quality of the solution increases proportional to the amount of time the algorithm executes. In addition, anytime algorithms always produce a result regardless of when they are interrupted. [2,3,4]

The multiple method approach does not rely upon continuous processing to solve a problem. Rather, a set of methods is available to solve a task. Each method has different characteristics that make it more or less appropriate given the current conditions. Every method solves the same problem, but varies in the amount of time it needs to produce the result and the quality of the result. There is a quality-time tradeoff between methods where a shorter execution time is achieved through reducing the quality of the result

[2,3]. Figure 1 [4] illustrates the major difference between an anytime algorithm and the multiple method approach.

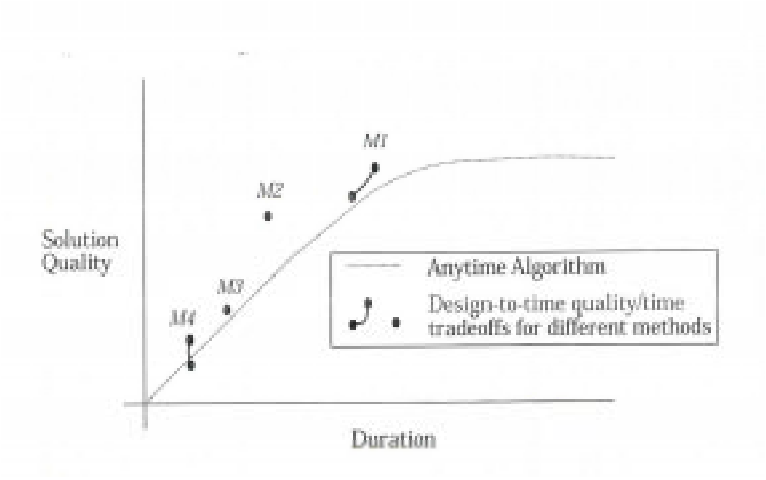


Figure 1: Anytime algorithm versus design-to-time tradeoffs.

Both approaches provide tremendous flexibility when determining an execution schedule. The advantage of using anytime algorithms is that they can fit into any available time slot whereas the multiple method approach allows for multiple, yet discrete (non-continuous), execution times. Garvey and Lesser [2] give two potential advantages of the multiple method approach over an anytime algorithm approach. The first is that it does not rely on the existence of iterative refinement algorithms that produce incrementally improving solutions as the runtime increases. Some problems may not have a solution that can be implemented by an anytime algorithm. A second advantage to the multiple methods approach is that the methods may be completely different approaches to solving the problem. These approaches can have very different characteristics depending on particular environmental conditions. An additional challenge

to the anytime algorithm approach is developing an algorithm whose performance is independent of environmental variables.

3.2 Definition for a Real-time Agent

A real-time agent (RT Agent) can be defined as an autonomous software entity that acts rationally under time-constrained conditions by means of real-time AI and a predictable well-defined "fallback method". The requirement of real-time AI provides the agent with the ability to make a quality-time tradeoff, either discretely or continuously, for any solutions it may produce. Below is a bulleted list explaining in greater detail the major points of the definition of an RT Agent.

- RTagents are autonomous (separate process/thread. refer to section 2.1).
- Solution methods are implemented using real-time AI techniques. An RTagent may have multiple methods for solving the problem(s) for which it was designed. Each method has a different execution time such that a longer execution time produces a better result. Alternatively, the solution may be generated through the use of an *anytime algorithm*.
- A RTagent is rational (refer to section 2.1).
- The RTagent decides based upon parameters passed to it which method is initially the most appropriate to execute.
- Every RTagent has a "fast fallback method." This is a default method that is executed under highly constrained conditions. This method's execution time

is well defined and predictable. It ensures that an agent can always provide an answer in a specified amount of time with some minimum amount of quality.

- Every method has a statistically derived "quality" value. This is an objective metric that can be used to compare the solution quality of different methods. How this value is determined shall be explored later.

3.3 Real-Time scheduling

As stated by Tanenbaum [6], real-time scheduling algorithms can be characterized by the following parameters

1. Hard real-time versus Soft real-time
2. Preemptive versus Non-preemptive scheduling
3. Dynamic versus Static
4. Centralized versus Decentralized

These algorithms attempt to schedule a set of tasks for either a single processor or multiple processors. They are most concerned with the timing constraints each task has associated with its execution. Each task will have a deadline before which it must be executed. Guarantees on meeting these timing constraints and how the system handles those tasks that cannot meet their deadline, differ based on which of the above characteristics the algorithm possesses. Hard real time defines those systems that require a 100% guarantee that all tasks meet their deadlines. Soft real time systems are more lax. In a hard real time system a task has a negative value if it exceeds its deadline. In

addition, it may even have catastrophic consequences. In a soft real time system the task still has value, although that value is reduced. Soft real time is generally characterized as an " as close as possible" approach.

Preemptive and non-preemptive algorithms differ in their handling of task execution. A preemptive algorithm has the ability to suspend a task that is currently being executed so a task of a greater priority can execute first. Non-preemptive scheduling does not have this ability so all tasks are executed to completion once started.

Dynamic and static algorithms are distinct based upon when they make decisions about scheduling. A dynamic algorithm makes these decisions "on the fly" during execution. Static algorithms make all scheduling decisions before run time. For example, these decisions may be stored in a table. When a task enters the system a table lookup is performed to see how the task should be scheduled.

A centralized system utilizes a single machine to collect information and to perform decision-making. In a decentralized system, decisions are made at the processor level.

While many different types of scheduling algorithms have been proposed, only the dynamic algorithms are of relevance to this project since RTMAS dynamic. A common dynamic scheduling algorithm is *earliest deadline first* (EDF) scheduling [8]. An EDF algorithm maintains a list of waiting tasks to be executed. This list is always sorted by deadline with the first task having the earliest deadline. When a new task enters the system, it is inserted into the list of waiting tasks. When system resources become free, the first task of the list is removed and executed.

3.4 Design-to-time Real-Time Scheduling

Design-to-time is an approach to problem solving in resource-constrained domains where: 1) multiple solution methods are available for tasks, 2) those solution methods make trade-offs in solution quality versus time, 3) and satisfying solutions are acceptable [2]. Design-to-time involves designing a solution to a problem that uses all available resources to maximize the solution quality within the available time. Design-to-time has some very specific characteristics:

- The domain may have both soft and hard real-time constraints
- As stated above, multiple solution methods are available to solve a specific task or set of specific tasks. These methods make a tradeoff between execution time and solution quality.
- Under highly constrained conditions, any solution that returns a result above a satisfying threshold is acceptable. The concept of a satisfying threshold implies that in general it is different than the optimal solution.
- The predictability of resources and deadlines is reasonable.

Design-to-time is a problem solving methodology of the type described by D'Ambrosio. It is a methodology that "given a time bound, dynamically constructs and executes a problem solving procedure which will (probably) produce a reasonable answer within (approximately) the time available"[1,4]. Bonissone and Halverson were the first to use the term "design-to-time" to refer to these systems defined by D'Ambrosio. Design-to-time real-time scheduling was defined by Garvey and Lesser at the University of

Massachusetts. Design-to-time real-time scheduling utilizes the "Multiple Method" implementation versus "anytime algorithms". When it is necessary for the system to adjust resource allocation, it can either postpone a task's execution or it can change the current problem solving method in the pursuit of optimal schedulability. This is referred to as approximate processing. Each method has a different change in quality for a one time-unit change in execution time. As long as the solution quality of each task remains above the aforementioned quality threshold, a modification to the task's problem solving method is allowed.

4.0 Real-Time Multi-Agent Systems

4.1 An Example Real-Time Multi-Agent Application

While current multi-agent systems can provide the underlying software to build autonomous agents, they do not have the capacity to express or enforce timing constraints on actions [5]. A pilot training simulation for a commercial airline is an example of an application for which a real-time multi-agent system (RTMAS) would be useful. Imagine a human pilot flying, landing, and taking off in a virtual environment with numerous other virtual planes and virtual weather conditions. These other planes and weather conditions (snow, rain, heavy winds) could be implemented as agents, but it is obvious that unless the system and the agents themselves can respond in real time, real world effects can not be accurately simulated, and the application is of little value as a training tool. Some of the timing constraints that would be placed upon the agents'

reactions to environmental stimuli could be due to the movements of other planes in the simulation, detection of possible collisions, or the time delay of the effect of a specific weather pattern. These timing constraints must be expressed in a formal way in the system, and there must be mechanisms to enforce them.

It is interesting to note that the proposed system can be used to design both real-time and non real-time systems. This provides a tremendous boost to the extensibility of such systems (i.e. theoretically, a real-time application can be converted to an application that doesn't require the enforcement of timing constraints by setting all the deadlines to infinity). A multi-agent system may not have real-time constraints associated with it, but it may be a requirement for the system sometime in the future due to an ever changing computing environment or a shift in demand.

4.2 Monitoring Agents and Cognitive Psychology

A significant amount of agent research has involved agents that monitor their environment and perform a particular action(s) in response to an event. While the RTMAS scheduling algorithm is applicable to any RTMAS during periods of duress, the concept developed from research in systems that utilized "Monitoring Agents".
A "Monitoring Agent" as defined here is an agent that monitors an environment and chooses an appropriate action from a set of unique Event $\rightarrow$ Action mappings. To this end the agent is in a perpetual cycle of *collecting* data (monitoring the environment) and *analyzing* data (reacting to events in the environment). This set of mappings need not be static. The distinction between a dynamic and static set of mappings is a major component in determining if an agent has the ability to learn. This is similar to how a

human being interacts with its environment. The "human monitoring agents" are our five senses; the eyes, ears, nose, mouth, and skin collect environmental data, process the data, and send that information to the brain for additional processing and action/reaction. Psychology uses the terms *perception* and *cognition* respectively for the concepts of collecting and analyzing data. Thus these terms can be applied to agents as well. One can speak of the degree of perception or cognition embodied by a software agent. It is the cognitive ability to hypothesize, estimate, and guess that prove vital in humanity's unique manipulation of its environment. Thus, these are also key elements in determining how capable a software agent is.

The predecessors to our RTMAS scheduling algorithm were developed as part of systems incorporating "monitoring agents". Garvey and Lesser [1] describe a "network of vehicle monitoring nodes." Each node analyzes acoustically sensed data in an effort to locate and monitor vehicles moving through a two dimensional space. The vehicles in their research are fish, ducks, and pigeons. The agents' must locate ducks and track their movement, attempting to detect as soon as possible when a duck plans to attack a fish. The agent must then warn the fish that it is in danger, allowing the fish enough time to escape. Pigeons in this case are not viewed as a threat to the fish, but still must be detected by the system and determined to be harmless.

Soto,Garijo,Iglesias, and Ramos conducted their research using a simulated robot and titled their system AMSIA. The robot is randomly assigned a collection of missions to accomplish. Each mission possesses a deadline, an importance, and a destination. A mission requires the robot to move to the mission destination and run a series of tests that can only be performed in that location. The robot must perform as many missions as

possible, with each mission being completed before the specified mission deadline. In addition, a number of different obstacles are randomly placed in the robot's environment.

Both research groups were concerned with maximizing system performance and maximizing the number of completed tasks during periods of stress. For Garvey and Lesser, such a situation occurs when a large group of pigeons appear in an environment where a flock of ducks is already being monitored. The system must analyze the type of each new vehicle and the relative risk that each vehicle poses to the fish, while still monitoring the patterns of previously detected dangerous vehicles. In AMSIA, such conditions occur when the robot is analyzing the proper method for completing a mission while still having to compute paths to avoid newly detected obstacles.

The means to maximize system performance and completed system tasks are identical to how psychologists understand and model human cognition. Although neither research group explicitly references these psychological concepts, they both describe identical functionality within their systems.

The first concept is the Perception-Action cycle defined by Ulric Neisser in his book "Cognition and Reality: Principles and Implications of Cognitive Psychology." The figure below shows this cycle.

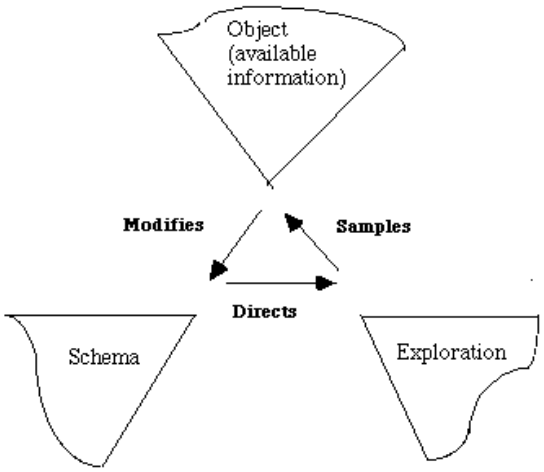


Figure 2: "The Perpetual Cycle", taken from Ulric Neisser.

Neisser developed it as a way of modeling how humans process information. He viewed it as a cycle that keeps operating as long as we are awake. We receive information from our environment (object / events), interpret it in terms of our knowledge (schema) and then carry out actions (exploration). Although simple, it is also a very powerful way of looking at the perception-cognition-action relationship. Both of the systems described earlier in this section behave in the same fashion. Garvey and Lesser define a sensor cycle as the period between two data retrieval actions; thus their cycle also consists of perception (identify new vehicles to the environment), cognition (reason about the vehicles), and action phases (continue to monitor threatening vehicles or warn a fish of an impending attack). The system is a series of the cycles over the period the system is active, just as Neisser proposes for humans.

The AMSIA system is defined as event driven. "Events are abstractions that the architecture uses to signal opportunities of activity, and they are stored in the events blackboard." Other components within the system monitor the events blackboard searching for events that they are programmed to analyze, and then compute the proper response(s). Proper responses are determined by searching appropriate "knowledge sources" (KSs). Neisser defines these knowledge sources as "schema". Humans, and any entity with the ability to learn, are also able to modify these schema based upon previous experiences.

The second concept was introduced by Kroemer in "Ergonomics: How to Design for Ease & Efficiency." He compared human behavior to a tank of full of gas. The gas represented a person's cognitive resources. The amount of cognitive resources available is finite. Thus without enough gas certain tasks cannot be completed. Kroemer was interested in extending the model of basic human information processing to account for times of stress or overload. Where Kroemer was interested in modeling periods of task demand exceeding available resources in human behavior, the design-to-time research and the AMSIA research were concerned with the same conditions for real time computing systems. Although Kroemer and Neisser may not have used computer science terminology in describing their work, they too were conducting research in the field of real time computing.

Similarities are also found in how these three systems, as well as this research, act under conditions of extreme duress. Since there exist only a finite amount of resources, a prioritizing of tasks must occur to determine which are executed and which are not. The human subconscious performs this prioritizing automatically. It can instantly assign an

importance to any bodily function allowing the brain to process only those tasks that are relevant to escaping the current danger. For example, when a person encounters a situation that is extremely life threatening (trapped in a fire, encountering a vicious animal) the brain immediately shuts down those bodily functions not necessary in responding to the current situation; digestion is just one example. Other functions may just have the portion of allocated resources decreased and continue operating in a diminished fashion. Real-time computing systems also have the ability to shed tasks completely or, through approximate processing, achieve the best possible partial results with reduced computational time in an attempt to reduce resource overload. The objective is to modify the set of tasks in such a way that the total demand of the tasks is now lower or equal to the amount of resources, allowing the system to continue functioning. To this end, the concept of importance, when applied to a task, is a key component in real time computing.

The correlations between real time computing research and human cognition offered in this section illustrate the importance of this body of work to Human-Cognitive modeling with computers and artificial intelligence. Viewing the human body as a RTMAS can provide real-time computing researchers with a very powerful model and a set of solutions for problems that are currently being tackled in computer science. Thus the heuristic developed here can prove beneficial not only to the field of traditional real-time computing, but also to the field of artificial intelligence and robotics.

5.0 Quality

Quality is a very difficult term to quantify, especially in computer science. The difficulty lies in the inherent subjectivity of its application. An item or action may be of a high quality to person A but of a very low quality to person B based solely upon personality and taste. How can one define a universal metric that would allow an objective comparison to be made between the qualities of two solutions? The term "quality" is defined for this paper as the variance between the returned result and the optimal result for a given task or problem. This research is not concerned with defining an "optimal result". This is a very case-specific definition. For example, when doing a search on Altavista an optimal result can be defined as all links returned having a 90% or higher accuracy ratio with the entered query. When one is checking stock quotes online, the optimal result may be defined as all quotes being no more than one minute old. These optimal results are very specific to the requests and reject universal classification. Either the developer or system administrator must define them.

The scheduling algorithm we have developed is dependent upon the system's ability to quantify the result of a task and to compare the "quality" of the results for any two tasks. With the notion of a pre-defined "optimal result" for a problem, the variance between the actual result and the optimal result can be used as an objective metric to statistically compare the quality of different methods. It can also be used to analytically compare the quality of different results. Thus a search on Altavista with a 75% accuracy ratio (optimal result is 90%) has a variance of 17% where quotes delayed for two minutes (optimal result is one minute) would have a variance of 50%. It is important to note that

it is now the “optimal result” that embodies the subjectivity. An optimal result for a specific problem could vary from system to system.

6.0 System Models

6.1 Real-Time Agent Model

This section offers formal descriptions of both the Real-Time Agent model and the model for Real-Time Scheduling with Load Reduction. An example agent labeled “StockAgent” is used to motivate the explanations. The “StockAgent” is an agent designed to provide useful information to an investor regarding their portfolio such as current quotes or market trends.

RTagent = {S1, S2, …, Sn}

A RTagent is comprised of a set of *solvables* {S1,S2,…,Sn}. A solvable is a problem that the agent is programmed with the ability to solve. A particular agent theoretically may have any number of solvables, although from a design perspective an agent’s purpose should be focused and it solvables should be related. An agent should not contain a solvable for filtering data and calculating the y intercept of a function.

Solvable (S) = <O, ES >

For every solvable an *optimal result* (O) must be defined. The optimal result for a solvable may vary from environment to environment depending upon the developer, the

user, and the intended use of the agent. This is an objective, system specific definition of what is considered to be the absolute best result for this problem. Given a solvable that returns the price of an item at various retail stores, the optimal result could be determined as returning the prices from at least ten different stores. The optimal result for a solvable that returns the position of objects in an environment could be the location of all objects with a 5mm variance. The optimal result is used to calculate the quality of an execution strategy. **ES = {es1, es2, …, esn}** In addition, a solvable is comprised of a *set of execution strategies* (ES)**.**

Execution strategy (es) = <ex, q, tv>

An *execution strategy* (es) is a method or algorithm for computing a solution to the problem defined by the solvable. An execution strategy is comprised of three elements. The *execution time* (ex) is how long it takes a strategy to run. This time can vary from machine to machine due to available resources, hardware, or network congestion, so it must be set specific to each environment. The level of *quality* (q) is a rating of the result of an execution strategy. Quality is calculated as a percentage of the optimal result such that q = (strategy result / optimal result). This definition for quality is conditional upon the ability to quantify the result of a task. For example, if the optimal result is pricing of a particular Sony television from ten stores, and a strategy is only able to return prices from seven stores, then the quality for this strategy is 70.[1] The last component of an execution strategy is the *tradeoff value* (tv)**.** The tradeoff value is defined as the change

[1] A slightly different approach is necessary for tasks where the optimal result is lower than the results of the execution strategies. If the optimal result is a 5 second delay on stock quotes, what are the qualities of execution strategies that return 10 or 20 second delayed quotes? In this case all results should be 1/time.

in quality of two execution strategies divided by the change in time. More precisely, for any es_i:

$$tv_i = \frac{\frac{q_i - q_{i+1}}{q_i}}{ex_i - ex_{+1}}$$

For the execution strategy with the least execution time the tradeoff value is undefined. Our scheduling heuristic uses the tradeoff value to determine a prioritized ordering of tasks that are candidates for reduction. Thus for the "StockAgent" we have:

Get_Quote[2]**:** As the name states, when called this solvable returns the current stock price for the user's portfolio or may be used to obtain a quote for a specific stock. The Get_Quote solvable has an optimal result (GQ-O) and two execution strategies for obtaining stock quotes:

GQ-O: An up to the minute quote for the specified stock symbol.

GQ-es1: The agent can access a premium service with up to the minute quotes. There is a higher overhead in time for performing such a transaction due to the authentication needed to access the service.

GQ-es2: Using a public server this execution strategy can receive specific stock quotes that are delayed by 20 minutes.

Thus a 5 sec delay is .2, 10 is .1, 20 is .05. Thus a delay that is twice as long as the optimal result is now calculates to a quality of 50 percent three times as long is 30 percent, etc.

[2] Note that all execution strategies are listed in decreasing order of execution time.

Advise[2]**:** This solvable is a bit more advanced. The agent is constantly collecting and monitoring various pieces of financial information, both long and short term (i.e. S&P Index, Dow Jones Average, Nasdaq Average, interest rates, beta values, yields of bonds, GNP, engulfing patterns, etc.) Based on a wide range of information, statistical computations and calculated correlations the agent acts as a broker, dispensing financial advice. It can estimate the expected return and growth of a portfolio or provide buy/sell with varying levels of confidence. The Advise solvable has three execution strategies for dispensing stock advice:

AD-O: The optimal result provides advice using all available information.

AD-es1: Provide recommendations and analysis surveying all available information.

AD-es2: Provide recommendations and analysis surveying all data in only the most influential data sets such as beta values, the Dow Jones average, and trends in interest rates.

AD-es3: Provide recommendations and analysis surveying only half the data in the most influential data sets.

For the "StockAgent" solvable *Advise*, the optimal result is defined as a result with a 99% confidence interval calculated using all available information. Execution times would be obtained from a statistical average of the results from executing the strategy some statistically significant number times or simply could be equal to the worst-case execution time. Thus for the three execution strategies:

Optimal Result = 99% confidence interval using all available information

Execution Strategy	Execution time (ex)[3]	Quality (q)[4]	Tradeoff value (tv)
AD-es1	7ms	95	.0789
AD-es2	5ms	80	.0833
AD-es3	2ms	60	∞ - undefined

Table 3: Example tradeoff values.

Finally for the "StockAgent" we have:

$\mathbf{ES_{stockagent} = \{Get\_Quote\ ,\ Advise\ \}}$

$\mathbf{GQ\text{-}es1 = <ex_{1GetQuote},\ q_{1GetQuote},\ tv_{1GetQuote}>}$

The set of execution strategies for a solvable S can be referred to as ES_S. The components of an execution strategy $es_i \in ES_S$ can be referenced as ex_{is}, q_{is}, tv_{is}.

6.2 Scheduling under Load Reduction

This section provides definitions and requirements for a system that utilizes a load-reduction scheduling algorithm. When a client makes a request, the scheduler processes the request and checks its schedulability. If the task is schedulable, the request is sent to the server, otherwise the scheduling service attempts to reduce the load on the system. If the task is still not schedulable after the load reduction, then an exception is raised informing the client of the reason for the task not being executed.

[3] These values do not represent real calculations
[4] These values do not represent real calculations

6.2.1 Definitions

- Request (R) = <A, V, I, D, H>

  A is the name of the agent upon which the request is being made. The variable *V* represents the name of the solvable that is being requested. *I* is the importance of the request. It is conceivable for some systems that the client may not be able to provide the importance of its request. This value could be set by the application depending on the type of request or the requestor. *D* represents the deadline of the request. An answer must be returned to the client before this time. Lastly, the variable *H* represents the quality threshold of the request. A solution with a quality below this value is of no use to the client.

- Currently Scheduled Tasks (T) = $\{t_1, t_2, \ldots, t_m\}$

- Scheduled Task (t_i) = < A, V, I, D, H, CE, RE, C>

  A, V, I, D, H – the specifications of the request (R) as defined above

  C : The cost of a task is a calculated value to determine the relative "cost" to the system of reducing the quality of the task (by choosing the execution strategy with the next fastest execution time). Cost is defined as the current execution strategy's tradeoff value (tv) times the importance (I) of the task. If the current execution strategy has the lowest execution of all strategies for a solvable, then the cost is set to ∞. Refer to section 6.2.4 for further information regarding a task's cost.

CE – the current execution time of the solvable. Execution strategies are referenced by their execution times rather than string identifiers like the agent and solvable names.

RE - the remaining execution time of the task. This attribute is necessary for tasks that are pre-empted.

- Scheduler

  The scheduler is an object that receives client requests and makes decisions based on the various characteristics of a task. The scheduler has a number of responsibilities including task creation, schedulability analysis, calling the heuristic, and exception handling.

6.2.2 Scheduling

This section describes the scheduling algorithm in detail. Scheduling starts when a client makes a new request $R_c = <A_k, V_{k,j}, I_c, D_c, H_c>$ for the solvable $V_{k,j}$ in Agent A_k. The request is received by the scheduler and the task $t_{m+1} = <A_k, V_{k,j}, I_c, D_c, H_c, CE_{k,j,1}, RE_c, C_k >$ is created. Note that there are *m* tasks already scheduled before the request is made. The current execution time (CE) defaults to the execution time of the execution strategy with the highest quality. Ideally, one hopes the task can be scheduled at this quality. Lower quality strategies are only considered during periods of overload. The remaining execution time (RE) is equal to the current execution time (CE) since the task has not yet begun executing. Task t_{m+1} is then added to the set of currently scheduled tasks T. Next the scheduler performs an EDF schedulability analysis on all tasks in T. If all the tasks are schedulable, then the client is notified that the request is schedulable and

is sent a priority and an identifier for the request. The client then passes these as arguments in the call to the agent. If the set T is not schedulable, then the load reduction heuristic is performed. Copies of the tasks in set T are stored by deadline in a sorted list for the schedulability analysis. The schedulability analysis also determines a list of tasks that are candidates for reduction (RC). Note that the list of candidates is most often a small segment of the larger list of all tasks in the system. The heuristic is performed on this list of candidates. This allows the heuristic to make any changes necessary for analysis while maintaining the integrity of set T.

6.2.3 Selection of Reduction Candidates

The EDF schedulability analysis uses a task's deadline, a task's remaining execution time, and the time the schedulability analysis begins to determine if a set of tasks is schedulable. Starting at the head of the list, every task in the deadline-prioritized list is visited. Two temporary variables are used named *TotalEx* and *TimeToEx*. The value of $TotalEx_i$ is the total amount of time necessary to execute all the tasks up to and including t_i. The value of $TimeToEx_i$ is the amount of time available to execute all the tasks up to and including t_i, such that the tasks t_0 to t_i will meet their deadlines. When the *ith* task is visited, the sum of its remaining execution time and its blocking time are added to $TotalEx_i$ (blocking time is assumed to be zero throughout this research and is only discussed in Section 9.2 Future Work and Extensions). The variable $TimeToEx_i$ is assigned the difference between the start time of the analysis and the deadline of the *ith* task t_i. More precisely:

$$\text{TotalEx}_i = \sum_{k=0}^{i} \text{(remaining execution time of } t_k + \text{blocking time of } t_k)$$

$$\text{TimeToEx}_i = \text{start of sched analysis} - \text{deadline of } t_i$$

If for some t_i, $TotalEx_i$ is greater than $TimeToEx_i$, then t_i will not meet its deadline. In stricter terms, the non-schedulability condition for a task t_i is:

$$TotalEx_i > TimeToEx_i$$

A pointer named *EndOfList* is then set to t_i. The analysis continues until all the tasks in the list have been visited. If it is determined that some task t_{i+j} will also miss its deadline, then *EndofList* is reset to point to t_{i+j}. The pointer *EndofList* is then passed to the heuristic. The amount of time that must be freed to make the set of tasks schedulable is also passed to the heuristic. This value is calculated as:

$$TimeNeeded = TotalEx_{\text{EndOfList}} - TimeToEx_{\text{EndOfList}}$$

In turn, the heuristic creates a local list that contains copies of all the tasks in the schedulability analyzer's list, up to and including the task pointed to by *EndOfList*. Thus, the list maintained by the heuristic is simply a sub-list of the one used for the schedulability analysis. It is critical that the schedulability analysis visit all the tasks before any call to the heuristic is made. This reduces the number of calls made to the heuristic for a given set of tasks when multiple tasks will miss their deadlines. For example, assume a set of tasks $T = \{t_0 \ldots t_7\}$. The task t_6 will miss its deadline by 7 seconds and t_7 will miss its deadline by 10 seconds. Imagine the heuristic was called as

soon as it was discovered that t_6 would miss its deadline. If a solution exists so that the first six tasks are schedulable, there is no guarantee this solution frees enough time to allow t_7 to meet its deadline as well. The heuristic must be called again to find a suitable solution that allows all the tasks to meet their deadlines. If the schedulability analysis had visited all the tasks before calling the heuristic, the decision as to whether set T was schedulable could be made with one call to the heuristic (versus one call per task that will miss its deadline). Additionally, what if there does not exist a solution that allows all the tasks to be scheduled? Then the call to the heuristic for tasks 0 through 6 was made in vain. Processing time is a scarce commodity that cannot be wasted in a real time system, especially during the times of resource overload for which the heuristic is designed.

6.2.4 Cost

The concept of a task's cost is a critical part of how the heuristic performs load reduction. While the tradeoff value measures the relative reduction in quality from executing the next fastest strategy, cost is a heuristically driven value that measures the reduction in the overall system quality. The cost of a task is directly proportional to its importance. Reducing or shedding a task of high importance is more detrimental to the system than reducing a task of lesser importance. This stems directly from the definition of importance. The term importance implies a value judgment of the superior worth or influence of something or someone. Thus, altering the state of an entity with a high importance will have a greater negative impact on the environment than altering an entity of low importance.

The cost of reducing a task is calculated by multiplying the task's tradeoff value times the task's importance. In this way, tasks with equivalent costs are then prioritized by their importance. Since our goal is to develop a system that maximizes quality, under ideal conditions the heuristic can find a valid schedule by only reducing low importance tasks with a low tradeoff value. The tradeoff value measures the precise amount of quality lost, while the importance of a task calculates the impact on the system of that quality loss.

6.2.5 Load Reduction Heuristic

This section is dedicated to explaining the heuristic. It begins with a brief summary as to how the heuristic works, followed by the pseudo code for the heuristic and a detailed explanation of the code.

The heuristic takes a pointer to a node (task) in the schedulability analyzer's list. It makes a copy of that node and traverses the list backwards making copies of all the tasks previous to the task to which the pointer points. This local list is then sorted by each task's cost.

The heuristic attempts to reduce the task with the lowest cost. If possible, it does so, and recalculates the time needed. If not enough time has been freed, the cost of the reduced task is recalculated and the list is resorted. If the recalculation of the time needed equals zero, the task is schedulable and the heuristic stops executing. Alternatively, the heuristic ceases execution if the cost of the lowest cost task is infinity. A cost of infinity means that the current execution strategy for a task is that agent's fastest strategy for the given solvable. Such a task is only seen when the cost of every

task is infinity and indicates that no further reduction is possible. If enough time can be freed through load reduction, then a pointer to the heuristic's list is returned to the schedulability analysis. The execution strategies of the corresponding tasks used by the schedulability analysis are then updated to achieve schedulability.

```
1) //Copy nodes from list pointed to by ptr into local list
2) Copy(ptr from argument list);
3) //Sort list of reduction candidates (RC) by their cost (C) in descending
   //order.
4) Sort ( );
5) TimeNeeded = CalcTimeNeeded();
6) temp_ptr = first task in list; //the task with the lowest cost
7)
8) while (temp_ptr->Cost ! =  ∞ && TimeNeeded > 0)
9) {
10)//if quality of next execution strategy is < quality threshold of the
11)//task.
12)if  (temp_ptr->next quality (Q) < temp_ptr->quality threshold (H) )
13)   action = 1;
14)else
15){
16)   //if execution time of next execution strategy is < remaining
17)   //execution time of the current task, time can be freed by altering
18)   //the execution strategy.
```

```
19)   if  (temp_ptr->next_es.execution time < RE )
20)     action = 2;
21)   else
22)   {
23)     //check the execution times of other execution strategies
24)     if  (temp_ptr->some_es.execution time < RE)
25)        action = 2;
26)    else
27)        action = 1;
28)   }// end else
29)
30) switch(action)
31)  {
32)   case 1: temp_ptr->cost =  ∞;
33)               temp_ptr = temp_ptr->next;  goto 12;
34)               break;
35)   case 2: int NewEt = temp_ptr->new_exec_time;
36)               temp_ptr  = new task <S, I, D, H, NewEt, RE_new,Cost_recalc >;
37)              // RE_new = CE_least+1
38)              TimeNeeded =  CalculateTimeNeeded();
39)              break;
40)  }  //end switch
41)
42)  Sort( );
```

```
43) temp_ptr = first task in list; //the task with the lowest cost.
44)} //end while loop
45)
46)if  (TimeNeeded <= 0)
47)   return (pointer to local list);  //set of tasks in now schedulable.
48)else
49)   return NULL; //task is not schedulable.  Exception raised to client
```

Figure 3: Heuristic pseudo code.

The heuristic detailed above is a more general, more readable version of the implemented heuristic. To start, the list of reduction candidates is copied and sorted by cost (C), the variable *TimeNeeded* is set, and the temporary pointer is set to the head of the list (lines 1-6). The variable *temp_ptr* always points to the reduction candidate with the lowest cost. When a node is visited one of two conditions is checked. The first condition is whether or not reducing the task will violate the quality threshold specified by the client (line 12). If this is the case the cost of the task is set to infinity (line 32) because this task is not reducible. The heuristic moves to the next task (line 33) and begins again. If the quality threshold is not violated then the remaining execution time of the task is checked. The remaining execution time needs to be less than the execution time of the next strategy (line 19) for reduction to occur. The only time this is not the case is if the task to be reduced has already been partially executed. If the task can be reduced then the execution time is changed, the current execution time (CE) is updated, and the remaining execution time (RE) is set equal to CE (lines 35-37). The cost (C) of reducing the task is recalculated using the tradeoff value for the new execution strategy. *TimeNeeded* is then reset to the new amount of time that must be freed (line 38). This is

done by setting *TimeNeeded* to the return value of *CalcTimeNeeded*. *CalcTimeNeeded* starts at the beginning of the list calculating the amount of time it will take to execute every task at its current execution strategy. It returns the difference between this value and the amount of time available for the tasks to execute plus an estimate of the running time for the heuristic.

The list of candidates is then resorted (line 42), and the temporary pointer is reset to the new head of the list (line 43), which is the new least-cost node. If the remaining execution time is greater than the next execution strategy, the heuristic searches for an execution strategy with an execution time that is less than the task's remaining execution time (line 24). If such an execution strategy is found the appropriate changes are made (lines 35-37). Otherwise, the task should no longer be considered for reduction. Therefore the cost is set to infinity (line 32).

When the while loop finishes executing, the value of *TimeNeeded* is checked (line 46). If enough time has been freed, the heuristic returns a pointer to the head of its list so the schedulability analysis can update its tasks with the new execution times (line 47). Otherwise, NULL is returned to indicate the failure of the heuristic to determine a schedulable solution (line 49).

7.0 System Implementation

The scheduling algorithm described in Section 6 is implemented as a scheduling service that is part of a Real-Time Multi-Agent System prototype being developed at URI. This prototype is based on the RTMAS architecture depicted in Figure 3 as well as

other ongoing real-time dynamic scheduling work at URI. This research is the preliminary portion of the RT Agent Scheduling section of Figure 3 below. Section 7.2 describes in detail how the real-time agent scheduling algorithm from Section 6 is implemented. Section 7.3 defines a RT CORBA architecture and implementation for dynamic scheduling. It also describes the interface for the scheduling service that has been implemented for a RTMAS and the interface for the scheduling service for real-time systems used in previous URI research. The RTMAS scheduling service is an extension of the more generic real-time scheduling service for systems without software agents.

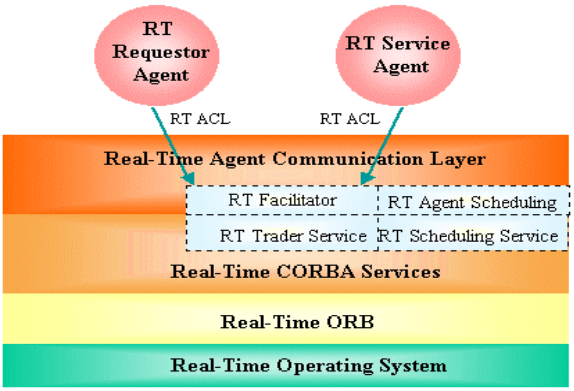


Figure 3: RTMAS architecture.

7.1 Determining the Correct Execution Strategy

This section explains how an agent knows which execution strategy to use for a call to one of its solvables. The following two sections will be easier to understand

provided this information. An agent is implemented as an object with its solvables being methods of the object. When the scheduling service determines that a client's request is schedulable, it passes back to the client the unique task identifier and an operating system priority at which the request will execute. The execution time associated with this task identifier is used to determine which execution strategy should be used. It is passed as a parameter in the client's call to the solvable. The agent's first action is to pass this identifier to the scheduling service and receive the execution time for the task.

Each solvable is implemented as a case-switch statement. The execution time that is returned to a solvable is used as the switch variable. The case statements execute a call to the execution strategy that has an execution time equal to the execution time received from the scheduling service. An example will help further to illustrate the process.

Client A sends a request for the *Get_Quote* solvable of the "StockAgent" defined in Section 6.1, to the scheduling service. This request is determined to be schedulable. The scheduling service returns a task identifier *x* to the client along with a priority *p*. The client then makes the call to the agent, passing *x* and *p* as a parameters. The solvable then passes x back to the scheduling service to receive the execution time *et*. The case statement that corresponds to the value of *et* is then executed in a thread running at priority *p*; specifically, a call to the execution strategy with an execution time of *et* is made. Note that depending upon a task's priority, a significant amount of processing time may elapse between the call to the solvable and the execution of the solvable. It is during this time that the heuristic may have reduced the tasks execution time. Thus, the decision as to which execution strategy to execute must be determined at the time of a solvable's execution.

7.2 RTMAS Scheduling Service Implementation

This section is dedicated to explaining in greater detail the workings of the RTMAS scheduling service. Figure 4 illustrates the interface of a scheduling service object as part of the research being conducted by the Real-Time Research Group at the University of Rhode Island. Figure 5 illustrates the interface for the derived scheduling service class used in the implementation of this research. Clients make requests by calling the *AddTask* method; agents advertise (register) with the scheduling service through the *AgentAdvert* method.

An agent is responsible for bundling all of its characteristics in an object that is a sub-class of the *Node* base-class. The Node class is an abstract class that defines the functionality common to all linked list elements in this implementation. The agent then passes only a pointer of type *Node* to the *AgentAdvert* function. Thus, if the agent characteristics that must be advertised change, the prototype of the *AgentAdvert* function is not affected. The *Node* pointer is then added to the A*gent Registry*, which is implemented as a linked list. The comments below the function in Figure 5 describe the members of the object currently passed to *AgentAdvert*. The function returns one (zero) if the insertion was completed successfully (unsuccessfully).

The *AddTask* function is designed in the same way as *AgentAdvert*. Again, the comments below the function describe in detail the members of the function parameter. When *AddTask* is called, the Agent Registry is immediately checked to ensure the "agent name-solvable name" pair is valid. If the pair is not valid, the function returns zero. A return value of zero raises an "invalid agent-solvable pair" exception to the

client. Next, the quality threshold of the request is validated. If the solvable does not have an execution strategy with a quality that satisfies the quality threshold indicated, then –1 is returned indicating a "threshold not met" exception, along with the highest quality the agent can provide.

Only after these two conditions are met is a new task created and added to the linked list *GlobalTable*. *GlobalTable* stores all the tasks in the system along with any information pertaining to a task such as the task id and deadline. The schedulability analysis object *Analyzer* is a derived class of *LinkedList*. A node in *Analyzer* merely contains a pointer to an entry in *GlobalTable* and an execution time indicating the current execution strategy chosen for this task. The nodes are sorted by deadline for the EDF analysis using the pointer to the *GlobalTable* node. The schedulability of the system is now checked, since the new request has been added, by a call to *CheckSched* in *Analyzer*. If the set of tasks is schedulable, an operating system priority and a task identifier are returned to the client, which are then passed as parameters in the call to the agent's solvable. A task's initial execution strategy is set to the strategy with the highest quality. If the schedulability analysis fails, then the heuristic is performed by a call to the heuristic object *LoadReducer*. If a satisfiable solution is not found, LoadReducer returns false and a "not schedulable" exception is raised with the client (*AddTask* returns three). In addition, the task is removed from *GobalTable* and the list of tasks used for the schedulability analysis. If the heuristic is successful, the flow of execution is the same as if the task were initially schedulable.

```
class SchedulingService  {
  public:
       SchedulingService();
       virtual ~SchedulingService();
```

```
      int AddTask(Node*);
     //int deadline, int importance, int threshold,

  protected:
     LinkedList     GlobalTable;
     SchedAnalysis  Analyzer;
};
```

Figure 4: IDL for the abstract scheduling service class.

```
class RTMAS_SchedulingService : public SchedulingService {
  public:
      RTMAS_SchedulingService();
      virtual ~RTMAS_SchedulingService();

      int AddTask(Node*);
     //int deadline, int importance, int threshold,
     //char* agent_name, char* solvable_name

      int AgentAdvert(Node*);
     //char* agent_name, char* solvable_name, LinkedList ExecStrats
     //a node in the execstrat list contains a tradeoff value (int),
     //a quality (int), and an execution time (int).


  protected:
     // LinkedList     GlobalTable; **Inherited**
     // SchedAnalysis  Analyzer;    **Inherited**

     Heuristic      LoadReducer;
     LinkedList     AgentRegistry;

};
```

Figure 5: Scheduling service IDL for RTMAS.

7.3 RT CORBA Architecture and Implementation

This body of work is part of a larger initiative by the URI Real-Time Research Group to develop a scheduling service for RT CORBA ORBs. Figure 6 below illustrates the basic system architecture of the RTMAS. The Scheduling Service has been designed as a CORBA object that receives all client requests. First, all agents must register with

the scheduling service (1) as described in the previous section. Clients can make requests to all agents that have advertise with the scheduling service(2). If a request is found schedulable, the client receives a priority and a task identifier from the scheduling service (3), otherwise the heuristic called (4). Once a client has received a priority and a task id, it then proceeds to make a RT CORBA method call to the server side agent (5). If a task is not schedulable, then an exception is raised with client.

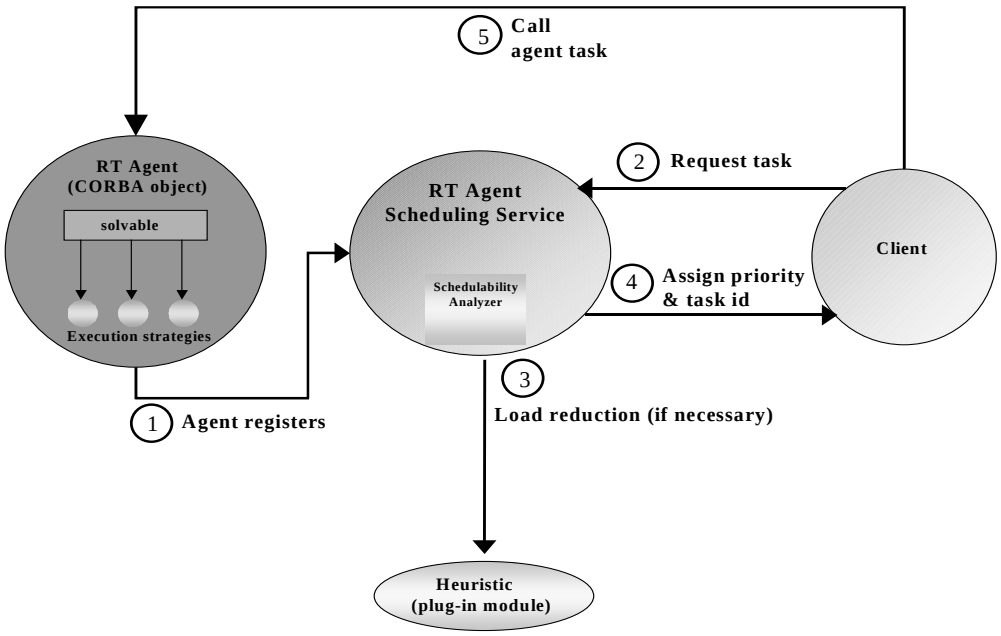


Figure 6: System architecture.

The RTMAS heuristic described in the previous section is designed to "plug in" to the scheduling service. It is one of a few such modules that are currently being developed within the URI RT research group.

There is one significant difference between the RT CORBA implementation and the implementation used for testing. Presently, when a solvable is executing, there is no

way to change the execution strategy being run. Significant work must be done to ensure that the precise remaining execution time of any task is always known for such a feature to be beneficial (for the current implementation the remaining execution time of a task is always equal to the initial execution time of that task). This functionality will be added through the use of signal handlers.

The future implementation in RT CORBA will also differ slightly in the assignment of priorities. Priority assignment for the current implementation is described in greater detail in Section 8. All testing was conducted using Linux, which only allows priorities to be set in the range of 1-99. There are a significantly greater number of RT CORBA priorities. Thus a mapping scheme will need to be developed for the RT CORBA system to run under Linux. The same is true of Solaris, which allows the user to set a task's priority in the range of 1-59.

The combination of a small range of priorities and the fact that these priorities are static allow only tasks with one of 99 distinct deadlines to be handled by the scheduler concurrently. Note that there can be any number of tasks assigned a specific priority. A greater detailing of the problem and a proposed solution are presented in Section 8 with the explanation of the test bed.

8.0 Results

This section details the tests that were performed and their results. A driver was designed to simulate client requests. In addition a randomizer was developed to randomly generate all the values of a request: deadline, quality threshold, importance, agent name and solvable name. In addition, the randomizer generates characteristics for

the randomly generated agents. The agent name, solvable name, quality of the execution strategies, and the execution times of the agents are all determined and set by the randomizer. A slightly different class was designed for the three different types of agents. All were derived from an Agent base class. There are three types of agents, each differing only by the number of execution strategies they provide. Every agent has only one solvable with two, three, or four execution strategies. As mentioned earlier, the execution times and the qualities are randomly generated and are set using accessor functions of the agent objects. The range of the randomized values differs depending on the test and are defined in the next section.

A total of forty-five agents are created with fifteen agents created for each type. Once all the agents are created, they register with the scheduling service. Once the agent registering is complete, the driver begins to create requests. Two command line options are provided. The first indicates the type of scheduling algorithm that should be used: Earliest Deadline First, Admission Control, or the Load Reduction Heuristic with an EDF schedulability analysis. For Admission Control, an EDF schedulability analysis is performed to determine if a request is scheduled. If it is not schedulable, the task is not executed and an exception is raised with the client. The second command line parameter indicates the number of requests to be made.

If a task is found to be schedulable, the priority and task id returned by the scheduling service is captured. The driver then spawns a thread at the given priority that executes the call to the scheduled task's solvable. Finally, after all requests have been processed and all scheduled tasks have been executed, the driver writes the results to file.

All the tests were run on Dell Pentium II 350 MHz machine running Red Hat Linux release 5.2 (kernel 2.0.35). The scheduling was implemented with the sched_setscheduler method, which is part of sched.h (exhaustive information can be found in the Linux programmer's manual). The user is provided with two real-time scheduling alternatives to the default policy: SCHED_FIFO and SCHED_RR. The driver spawns threads using the SCHED_FIFO scheduling algorithm because it does not use time slicing; thus it is the only algorithm of the two that allows the user to run processes following EDF.

A thread can be assigned a priority in the range of 1 – 99. Any number of threads can be given the same priority. As stated above, threads of the same priority execute in a first-in first-out order. Tasks are assigned a priority using the task's deadline and the time the driver begins. The difference between the tasks deadline and the system start time is subtracted from 99 to determine a task's priority. For example, if the system begins at 962457703 (seconds since 00:00:00 UTC, January 1, 1970) and a tasks deadline is 962457717, the task is assigned a priority of 85 (99 – 14). Unfortunately, tasks must be assigned static priorities. Therefore, the system encounters a problem when a task with a deadline greater than or equal to 99 seconds after the system start time needs to be executed. Priorities below one are not allowed and setting the priority to anything higher will violate EDF, since any tasks with deadlines earlier will be preempted. Therefore when the system has been operating for 99 seconds, tasks are not immediately spawned as threads once they are determined schedulable. Rather, the tasks are queued. When the last thread of priority 1 has been executed, the queued tasks are assigned priorities and executed. The priority assignment begins again from 99. The system knows the start

time, execution time, and therefore the expected completion time of each task. Thus, it is able to track when the priority assignment and execution of the queued tasks should begin. This is not a problem with systems that allow for dynamic priority setting. For dynamic priorities, the concept of "priority aging"[9] can be applied. The URI Real-Time Research Group developed the concept of priority aging which assigns a dynamic priority based upon how long a task has been in the system.

8.1 Tests

In all there were six different test suites. For the Baseline, Short Deadline, and Long Deadline suites, there are three different tests, each one utilizing a different scheduling algorithm: EDF, Admission Control, or RTMAS Heuristic. For EDF, requests were not checked to be schedulable before being assigned a priority and executed. They were simply executed. Admission Control performed an EDF schedulability analysis for each request. If a request was determined to not be schedulable it was not executed. The heuristic also performed an EDF schedulability analysis for each request. The difference being that for non-schedulable requests, the heuristic attempts to make the task schedulable through load reduction. If the request was found to still not be schedulable it was not executed. Each of the three scheduling algorithms was tested with twenty, forty, and sixty requests. Table 4 through Table 6 show the range of values randomly generated for each test suite.

The other three suites were designed to test the effect of the number of execution strategies defined for a solvable. They each used the same randomly generated values listed in Table 4. All the agents used in each suite had the same number of execution

strategies per solvable. One suite utilized agents with only two execution strategies per solvable, another utilized agents with only three execution strategies, and the last only agents with four execution strategies.

For all tests, the estimated processing time of the heuristic was set to two seconds. This amount of time was used by the heuristic when it calculated how much time must be freed to obtain a schedulable set of tasks. The overhead of the heuristic ranges from 20% to 100% of a task's execution, depending on the current execution time of the task to be scheduled.

Randomly Generated Value	***Low***	***High***
Deadline of request[5]	2	10
Importance of request	1	10
Quality threshold of request	50	90
Execution times of execution strategies	1	10
Quality of execution strategies	70	100

Table 4: Values for "Baseline Suite" and the "Execution Strategy Suite."

Randomly Generated Value	***Low***	***High***
Deadline of request[5]	1	3
Importance of request	1	10
Quality threshold of request	50	90
Execution times of execution strategies	1	10

[5]Deadline = Chart Value + Time Request Made + Longest Execution Time of Solvable

Quality of execution strategies	70	100

Table 5: Values for the "Short Deadline Suite."

Randomly Generated Value	***Low***	***High***
Deadline of request[5]	10	15
Importance of request	1	10
Quality threshold of request	50	90
Execution times of execution strategies	1	10
Quality of execution strategies	70	100

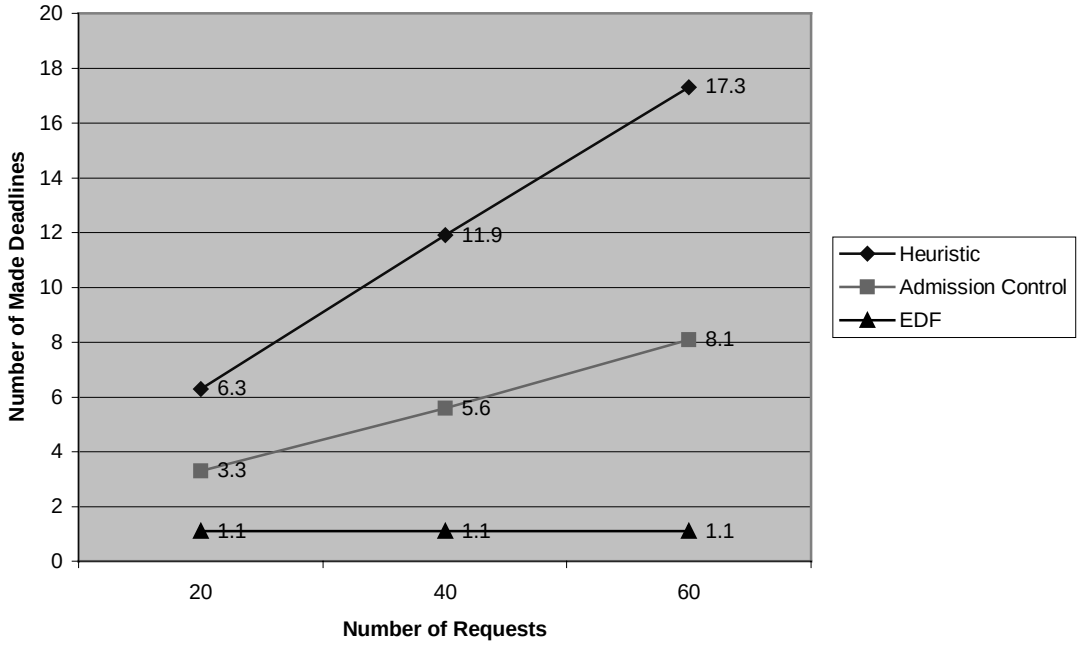
Table 6: Values for the "Long Deadline Suite."

8.2 Test Results

8.2.1 Made Deadlines

The primary metric used for comparing the performance of the three scheduling algorithms is the number of made deadlines. Figures 7, 8, and 9 plot the total number of executed tasks that met their deadlines. The tests were designed to simulate periods of extremely high resource contention. It is plain to see that scheduling with our heuristic allows a significantly greater number of tasks to execute. Also notice that as the number of requests increases, the difference between the number of successfully executed tasks

using the heuristic and the number of successfully executed tasks using admission control also increases. The slope for the heuristic's results is always greater than the slope of the plotted results for admission control, regardless of the length of the deadlines. It is not surprising to see such a low value for EDF since the algorithm is known to perform very poorly in periods of high duress.



.

Figure 7: Comparison of Successfully Completed Tasks for Baseline Suite.

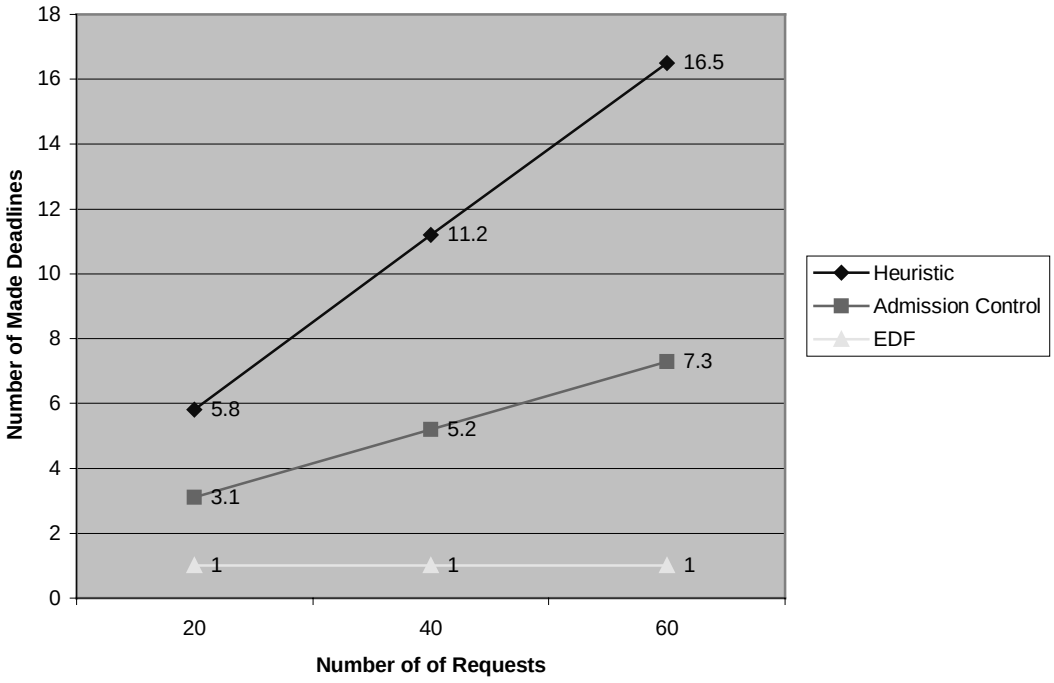


Figure 8: Comparison of Successfully Completed Tasks for Short Deadline Suite.

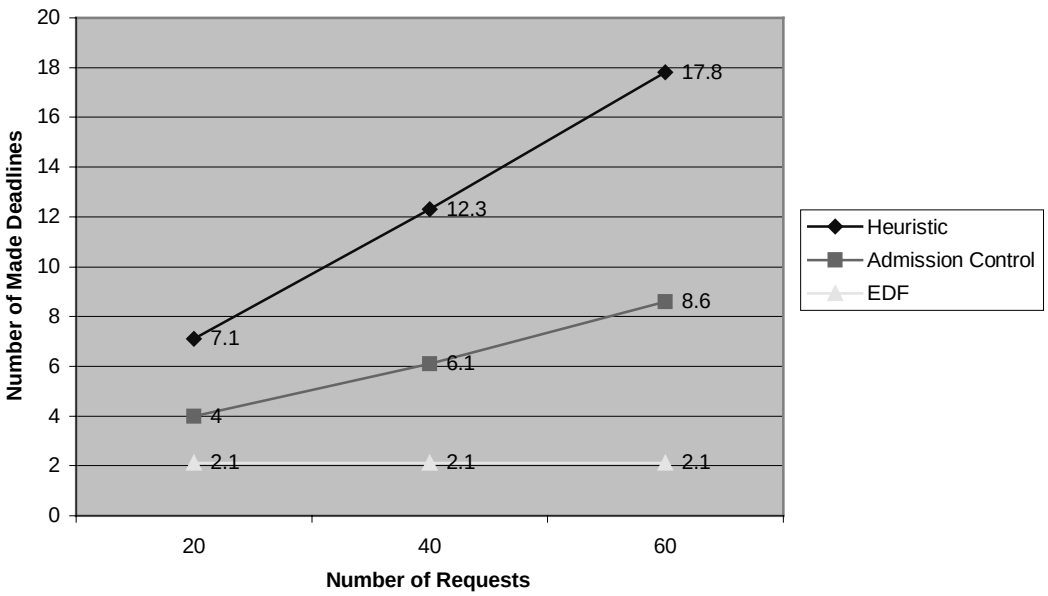


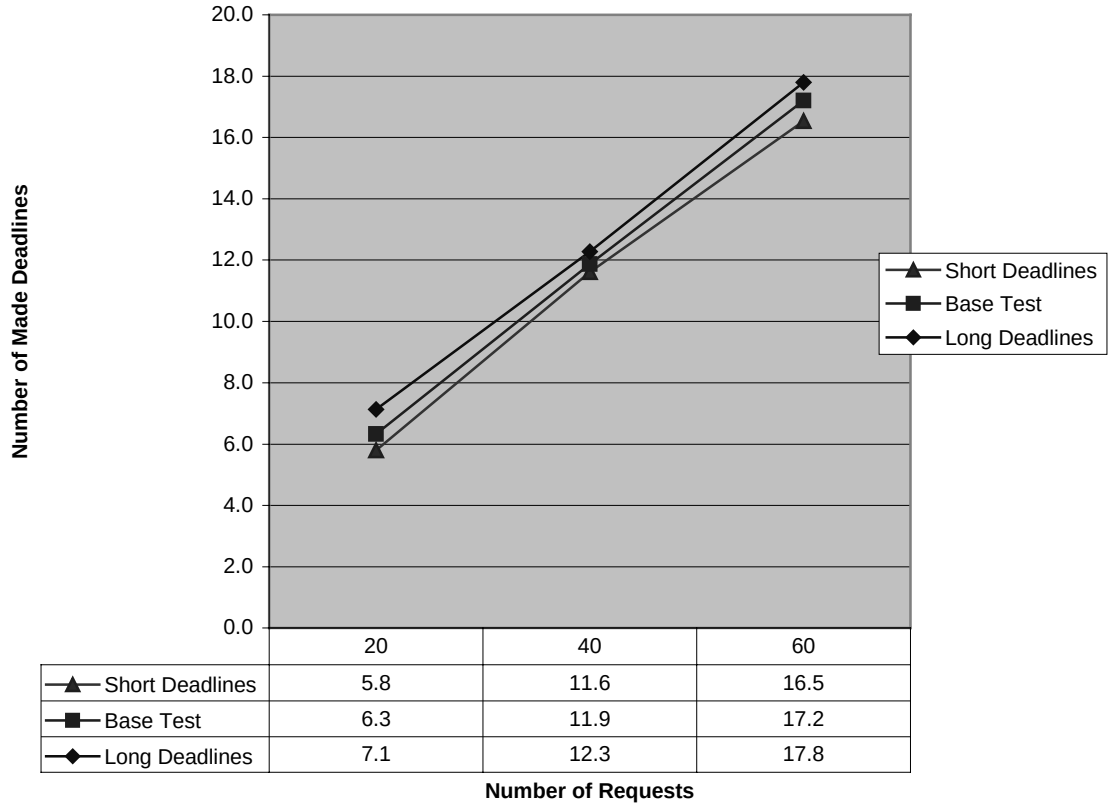
Figure 9: Comparison of Successfully Completed Tasks for Long Deadline Suite.

8.2.2 Effect of Deadline Length on Heuristic Performance

Figure 10 compares the performance of the heuristic for different deadline lengths. The length of the deadlines had a nearly negligible effect upon the total number of successfully executed tasks for the heuristic and admission control. EDF was able to schedule one additional task, for a total of two, when all tasks had long deadlines.

On average, for a set of requests with very long deadlines, our heuristic will be able to successfully schedule 1.3 more tasks than if the same set were comprised of requests with very short deadlines. Such a tight variance indicates that quality threshold is a significant contributing factor of the number tasks that meet their deadlines. Twenty tasks with long deadlines, as presented below, are easily schedulable if there exists no limit to how low the quality of the tasks can be reduced. All twenty tasks could

theoretically be set to run at their lowest quality, and consequently, at very low execution times. This would allow nearly all, if not all, the tasks to be scheduled. Yet, the quality threshold of a task prohibits such an occurrence. Even if all requests specify long deadlines, when a system is flooded resource contention will occur. If quality thresholds are all very high, most of the benefit to schedulability from the long deadlines is lost.



	20	40	60
Short Deadlines	5.8	11.6	16.5
Base Test	6.3	11.9	17.2
Long Deadlines	7.1	12.3	17.8

Figure 10: Made Deadlines using the Heuristic for Different Deadline Lengths.

8.2.3 Average Quality

Figure 11 is very important, for it illustrates the sacrifice that must be made to improve the number of schedulable tasks. The results are the same regardless of the length of the deadlines. The average task quality is reduced 11% by our heuristic for an approximate 50% increase over admission control in the number of successfully executed tasks (refer to Figures 7 through 9).

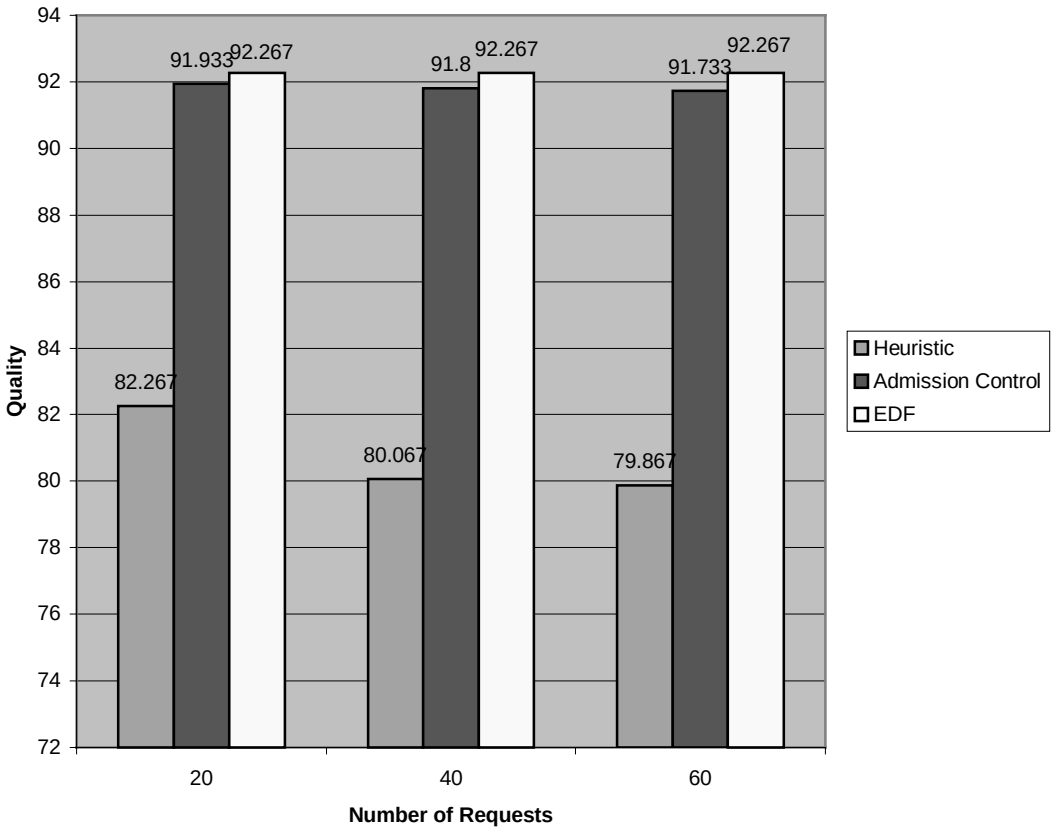
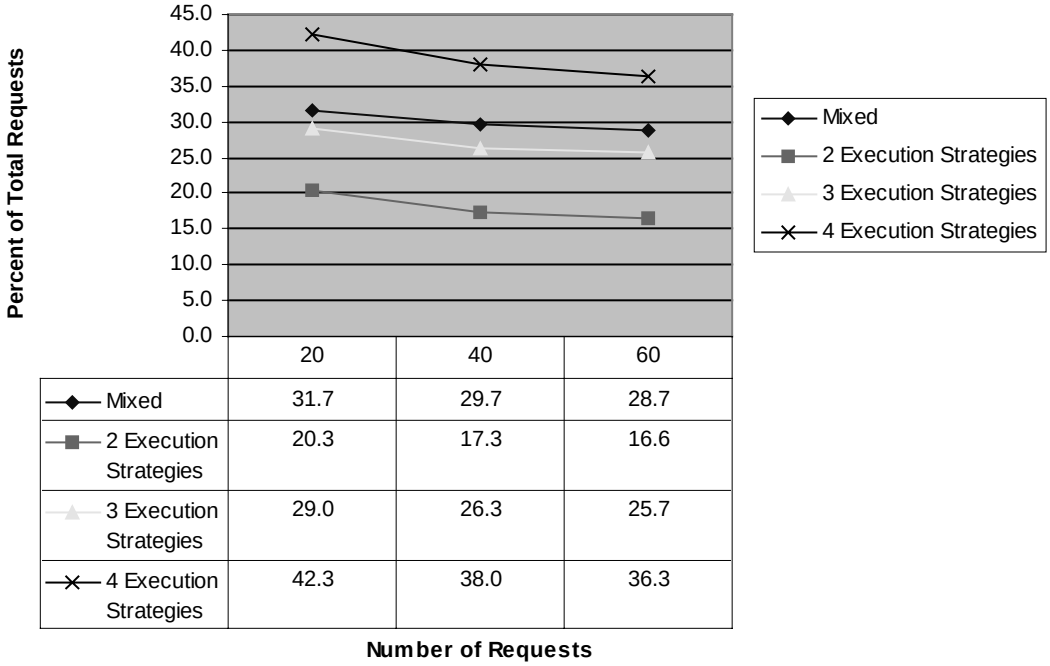


Figure 11: Average Quality Per Task.

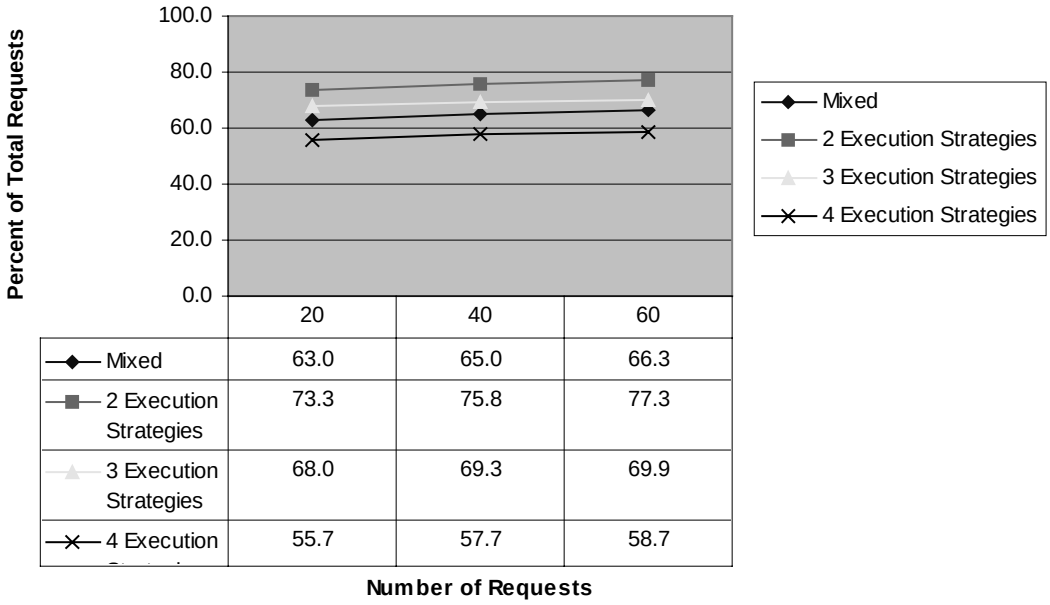
8.2.4 Effect of Execution Strategy Quantity on Heuristic Performance

Figures 12 and 13 clearly show the correlation between the heuristic's performance and the number of execution strategies provided for each solvable. The number of scheduled tasks that meet their deadlines is directly proportional to the number of execution strategies as illustrated by Figure 12. For a given number of requests, doubling the number of execution strategies doubles the percentage of made deadlines. This effect can be diminished though if the quality of the execution strategies is relatively low. As discussed in Section 8.2.2, the performance of the heuristic is very sensitive to the quality thresholds of the requests. Secondary execution strategies of significantly low quality reduce the positive effects of increasing the number of execution strategies by inhibiting the number of reduction candidates. Figure 13 illustrates the effect by focusing on the percentage of the requests that are not schedulable. Notice as the number of execution strategies increase, the slope of the plotted values decreases. Thus a system comprised of agents with four execution strategies shows less of a performance decrease as the number of requests increases versus systems of agents with fewer execution strategies per solvable.



	20	40	60
Mixed	31.7	29.7	28.7
2 Execution Strategies	20.3	17.3	16.6
3 Execution Strategies	29.0	26.3	25.7
4 Execution Strategies	42.3	38.0	36.3

Figure 12: Comparison of Made Deadlines for Execution Strategies Suite



	20	40	60
Mixed	63.0	65.0	66.3
2 Execution Strategies	73.3	75.8	77.3
3 Execution Strategies	68.0	69.3	69.9
4 Execution	55.7	57.7	58.7

Figure 13: Percentage of Tasks Not Schedulable

8.2.5 Additional Heuristic Statistics

The tables in this section are for the Baseline suite. The tables provide approximate base averages for the percentage of successfully executed tasks, tasks schedulable due to reduction, and tasks not scheduled. These averages can be used as reference points when trying to estimate the performance of other systems. Thus, about 30% of all requests are schedulable and about 22% of those are due to the heuristic. Approximately 65% of requests will not be schedulable during periods of high resource contention. Thus, from the previous section one can expect these percentages to increase if the minimum number of execution strategies for a solvable is three or greater. They can be expected to decrease otherwise.

Number of Tasks	***Short***	***Baseline***	***Long***
20	29%	31.6%	35.67%
40	29%	29.6%	30.67%
60	27.56%	28.69%	29.67%

Table 7: Percentage of Tasks Making Their Deadline

Number of Tasks	***Short***	***Baseline***	***Long***
20	23.3%	22.3%	25%
40	25%	22.5%	24%
60	22.89%	21.6%	24.2%

Table 8: Percentage of Tasks Schedulable Due to the Heuristic

Number of Tasks	*Short*	*Baseline*	*Long*
20	65.67%	63%	59%
40	65.5%	65%	64%
60	67.2%	66.29%	65.11%

Table 9: Percentage of Tasks not Schedulable[6]

9.0 Conclusions

This chapter summarizes the completed research and then discusses some key work to be done in the future.

9.1 Summary of Completed Work

While our research has answered many questions about real time scheduling with agents, some areas need to be researched further. It is important to formalize the research that has been done to clearly see what has been accomplished and what is the best way to proceed. Therefore to summarize, we have achieved the following at the end of this research:

1 Developed a model for Real-Time Agents.

2 Developed a heuristically driven scheduling algorithm that utilizes load reduction for a Real-Time Multi-Agent System.

3 Implemented and tested the algorithm in a simulated environment.

[6] Note: Percentage of tasks not executed due to quality threshold are not shown.

4 Substantiated the advantages and improved performance of such an algorithm over other scheduling algorithms.

5 Isolated key elements that can improve or detract from the algorithm's performance.

6 Developed benchmarks to help gauge the impact of future enhancements.

We were successful in achieving our goals as outlined in the introduction of this paper. The scheduling algorithm can provide CORBA developers the ability to schedule agents within a real-time environment. With the development of load reduction and the load reduction heuristic, we have also developed a means improving performance under periods of high resource contention. Although efficient algorithms already exist to do this, none are specifically designed with software agents in mind or to allow for load reduction. They are geared towards solving tradition problems in real-time scheduling while we have focused our work on real-time scheduling for future technologies.

While more work around the topic still exists, our research has provided an important foundation for future RTMAS research at URI. In addition, it provides a strong model for future work involving real time agents and a methodology for implementing load reduction.

9.2 Future Work and Extensions

All extensions and future work are listed below.

1 **Blocking time**

Blocking time will be added to the schedulability analysis. The blocking time of a task t is the amount of time left before which the task must start if it will make its deadline. The blocking time of a task t will be considered when a schedulability analysis will be done. In our implementation the blocking time of a task was assumed to be zero for all tasks.

2 **Remaining execution time**

The remaining execution time was always equal to the initial execution time. Indeed, before a task starts this is true. The remaining execution time is no longer equal to task's initial execution time when the task is preempted. The remaining execution time of a preempted task is the difference between its initial execution time and the amount of time it has already executed. This information will be used to determine of if it is quicker for a preempted task to finish executing the strategy it has already partially processed, or if it should start executing another strategy. It is also necessary to make the decision whether the currently executing task should be signaled to change the execution strategy it is running.

3 **Changing the execution strategy of currently executing task**

To allow a currently executing agent to change its execution strategy, each agent will define a signal handler function named *StratCheck*, of type void,

that takes a single integer argument. This function will retrieve the new execution time from the scheduling service and execute a **goto** statement that jumps to the beginning of the solvable function. From this point a call to the new execution strategy will be made. Each execution strategy must define a signal of the form **signal**(SIGALRM, *StratCheck*). When the scheduling service updates the execution strategy for a task(s), it must set an alarm to trigger the signal in the agent currently being executed. The code would be **alarm**(0) which causes the alarm clock to immediately go off and the signal SIGALRM is generated. When the signal is generated, control flow is switched from the execution strategy to the *StratCheck* function.

4 **Schedulability Analysis determines critical points**

The schedulability analysis will be enhanced to include the calculation of the time to be freed. This will help to improve the performance of the system since this amount can be determined at the same time it is determined if the set of tasks is schedulable. It will also improve the complexity of the heuristic by improving the time complexity of the calculation for the time that must be freed. Currently, the heuristic has a worst case time complexity of $O(n^k)$, where k equals the average number of reductions possible per task. This is very poor, but since the number of reduction candidates is generally below ten and k is strictly less than or equal to four in all the tests, there wasn't any effect upon the system performance. But these conditions cannot be guaranteed (nor should they be). Therefore the time complexity must be improved. By isolating critical points the complexity can be reduced to O(n).

The schedulability analysis will be modified to isolate critical points in the schedule. A critical point is defined as a point in the schedule where a task will miss its deadline. For each critical point, an amount of time is assigned; this is the amount of time that needs to be freed before the deadline of the task at the critical point. These critical points and their associated times will be passed to the heuristic.

5 Different Calculation for Cost

The formula for a task's cost could be changed to take into account other variables or to improve the performance of the algorithm. Currently, the cost calculation does not take into consideration the total amount of time that a reduction will free or the total amount of quality lost. Only importance and the percentage change in quality per time unit is used. For two tasks with the same costs, as defined within this paper, it may prove beneficial to always reduce the task that frees the most absolute time, has the lowest absolute reduction in quality, or some combination of both.

6 Alternative Base Scheduling Algorithm

It may prove beneficial to change the schedulability analysis to an algorithm that behaves better during times of high resource contention. For example, an algorithm that uses time slicing where the time quantum is a task's deadline. In this way, the foundation from which the load reduction begins can schedule more tasks naturally, and thus be modified by the heuristic to allow an even greater number of tasks to be scheduled.

10.0 References

[1] Huhns, Michael N., and Munindar P. Singh, ed. *Readings In Agents*. San Francisco: Morgan Kauffman Publishers, Inc., 1998.

[2] Garvey, Alan, and Victor Lesser. "A Survey of Research in Deliberative Real-Time Artificial Intelligence," *Journal of Real-Time Systems*, Vol. 6, No.3, May 1994.

[3] Garvey, Alan, and Victor Lesser. "Design to time Real-Time Scheduling." *COINS Technical Report 91-72*, University of Massachusetts, 1991.

[4] D'Ambrosio, B. "Resource bounded-agents in an uncertain world," *Proceedings of the Workshop on Real-Time Artificial Intelligence Problems*, (IJCAI-89, Detroit), Aug. 1989.

[5] Chorafas, Dimitris N. *Agent Technology Handbook.* Montreal: McGraw-Hill, 1998.

[6] Tanenbaum, Andrew S. *Distributed Operating Systems.* New Jersey: Prentice-Hall, Inc., 1995.

[7] Russell, S. and Norvig, P. *Artificial Intelligence: A Modern Approach.* New Jersey: Prentice-Hall, Inc., 1995.

[8] Liu, Jane. *Real-time systems*. New Jersey: Prentice-Hall, Inc., 2000.

[9] Fay Wolfe, Victor, Cingiser DiPippo, Lisa, Ginis, Roman, Squadrito, Michael, Wohlever, Steven, Zykh, Igor, Johnston, Russell. "Expressing and Enforcing Timing Constraints in a Dynamic Real-Time CORBA System," *Real-Time Systems Journal*, Vol. 16, No. 2-3, May 1999.

Bibliography

Bigus, Joseph P., and Jennifer Bigus. *Constructing Intelligent Agents in Java*. New York: John Wiley and Sons, Inc., 1998.

Chorafas, Dimitris N. *Agent Technology Handbook.* Montreal: McGraw-Hill, 1998.

D'Ambrosio, B. "Resource bounded-agents in an uncertain world," *Proceedings of the Workshop on Real-Time Artificial Intelligence Problems*, (IJCAI-89, Detroit), Aug. 1989.

Fay Wolfe, Victor, Cingiser DiPippo, Lisa, Ginis, Roman, Squadrito, Michael, Wohlever, Steven, Zykh, Igor, Johnston, Russell. "Expressing and Enforcing Timing Constraints in a Dynamic Real-Time CORBA System," *Real-Time Systems Journal*, Vol. 16, No. 2-3, May 1999.

Garvey, Alan, and Victor Lesser. "A survey of Research in Deliberative," *Real-Time Artificial Intelligence Journal of Real-Time Systems*, Vol.6, No.3, May 1994.

Garvey, Alan, and Victor Lesser. "Design to time Real-Time Scheduling." *COINS Technical Report 91-72*, University of Massachusetts, 1991.

Graham, John R. "A Proposal for the study of Real-Time Agent Scheduling for Distributed Agents." Newark: University of Delaware, 1999.

Huhns, Michael N., and Munindar P. Singh, ed. *Readings In Agents*. San Francisco: Morgan Kauffman Publishers, Inc., 1998.

Kroemer, Karl, Kroemer, Heinrike, and Kroemer-Elbert, Katrin. *Ergonomics: How to Design for Ease & Efficiency.* New Jersey: Prentice-Hall, Inc., 1994.

Liu, Jane. *Real-time systems*. New Jersey: Prentice-Hall, Inc., 2000.

Neisser, Ulrich. *Cognition and Reality: Principles and Implications of Human Psychology*. W. H. Freeman and Company, 1976.

Russell, S. and Norvig, P. *Artificial Intelligence: A Modern Approach.* New Jersey: Prentice-Hall, Inc., 1995.

Soto, Ignacio, Garijo, Mercedes, Iglesias Carlos A., Ramos Manuel. "An AgentArchitecutre to fulfill Real_Time Requirments," *Fourth International Conference on Autonomous Agents 2000.* Barcelona, Spain, June 2000.

Tanenbaum, Andrew S. *Distributed Operating Systems.* New Jersey: Prentice-Hall, Inc., 1995.

Toyn, Ian, Dix, Alan, and Runciman, Colin. "Performance Polymorphism." *Functional Programming Languages and Computer Architecture*, ed. Gilles Kahn, Springer-Verlag Lecture Notes in Computer Science 274, September 1987, pp. 325-346.

Webster's online dictionary at, http://www.m-w.com/dictionary.htm.

The scheduling algorithm is of paramount importance in a real-time system to ensure desired and predictable behavior of the system. Within computer science real-time systems are an important while often less known branch.Â A multiprocessor system will range from multi-core, essen-tially several uniprocessors in one processor, to several sep-arate uniprocessors controlling the same system. A distributed system will range from a geographically dis-persed system to several processors on the same board.Â A real-time system also have requirements based on dead-line, a real-time system can either be hard or soft depending on the consequences of missing a deadline. A hard real-time system is never allowed to miss a deadline because that can lead to complete failure of the system. Schedule Algorithm Problem Type Service Agent Call Center Problem Package. These keywords were added by machine and not by the authors. This process is experimental and the keywords may be updated as the learning algorithm improves.Â Journal of Intelligent Information Systems, Special Issue on Networked Information Discovery and Retrieva (1995)Google Scholar. 12. Nodine, M., Perry, B., Unruh, A.: Experience with the infosleuth agent archi- tecture. In: Proceedings of AAAI 1998 Workshop on Software Tools for Developing Agents (1998)Google Scholar. 13.Â Wang Y., Yang Q., Zhang Z. (2000) Real-Time Scheduling for Multi-agent Call Center Automation. In: Biundo S., Fox M. (eds) Recent Advances in AI Planning. ECP 1999. Real-time systems are systems that carry real-time tasks. These tasks need to be performed immediately with a certain degree of urgency. In particular, these tasks are related to control of certain events (or) reacting to them. Real-time tasks can be classified as hard real-time tasks and soft real-time tasks. A hard real-time task must be performed at a specified time which could otherwise lead to huge losses. In soft real-time tasks, a specified deadline can be missed. This is because the task can be rescheduled (or) can be completed after the specified timeÂ Least Slack Time (LST) scheduling Algorithm in real-time systems.Â Basic Model of a Real-time System. 01, May 20. Difference between Periodic and Aperiodic Real-time Tasks.