5-1-1999

Server selection for mobile agent migration

Wayne Caro

Follow this and additional works at: http://scholarworks.rit.edu/theses

Recommended Citation

This Thesis is brought to you for free and open access by the Thesis/Dissertation Collections at RIT Scholar Works. It has been accepted for inclusion in Theses by an authorized administrator of RIT Scholar Works. For more information, please contact ritescholarworks@rit.edu.
Server Selection for Mobile Agent Migration

by

Wayne Caro

A Thesis Submitted
in
Partial Fulfillment of the
Requirements for the Degree of
MASTER OF SCIENCE
in
Computer Engineering

Approved by:

Principle Advisor
Dr. Hans-Peter Bischof, Assistant Professor, Computer Science Department

Committee Member
Dr. Roy Czernikowski, Professor, Computer Engineering Department

Committee Member
Dr. David Suits, Associate Professor, College of Liberal Arts

Department of Computer Engineering
College of Engineering
Rochester Institute of Technology
Rochester, New York
May 1999
I, Wayne Caro, hereby grant permission to any individual or organization to reproduce this thesis in whole or in part for non-commercial and non-profit purposes only.

Wayne Caro

5/19/99

Date
Abstract

The purpose of this thesis is to develop, test, and simulate an algorithm that mobile software agents can use to select a server to which the agents can migrate. Software agents are autonomous software entities that perform tasks on behalf of other agents or humans, and that have some degree of intelligence. In particular, a mobile software agent is capable of migrating from one computer system (agent server) to another during the course of performing its tasks. Most current implementations of mobile software agents (simply referred to as agents) have simple forms of server selection. The algorithm discussed in this thesis proposes new ideas for dealing with the server selection process.

The algorithm proposed in this thesis is intended to provide a good basis from which further work can be continued in the area of agent server selection. This algorithm was demonstrated to work as expected under a set of boundary conditions of purely abstract computer resources. Then the algorithm was used in a simulation of a print job scheduler for a cluster of printers. Some of the concepts that this algorithm uses are resource importance factors, “needed” and “wanted” resources, risk factors, server resource evaluations, and server resource availability.
# Table of Contents

List of Figures ................................................................. iii
List of Tables ................................................................. iv
Glossary ................................................................. v

1 INTRODUCTION ........................................................................................................ 1
  1.1 Agent Description ................................................................. 1
  1.2 Mobile vs. Non-Mobile Agents ............................................... 4
  1.3 Trust and Security in Multiple Agent Systems ...................... 7
  1.4 Resource Allocation and Usage in Multiple Agent Systems ...... 9
  1.5 Conclusion of Introduction .................................................. 10

2 DESCRIPTION OF THE PROBLEM ................................................................. 11
  2.1 The Focus of the Thesis ......................................................... 11
  2.2 Assumptions ........................................................................... 12

3 OVERVIEW OF THE SOLUTION ............................................................... 15
  3.1 Needs and Wants ................................................................. 13
  3.2 Evaluations ............................................................................ 14
  3.3 Risk ...................................................................................... 15

4 DESCRIPTION OF THE ALGORITHM ......................................................... 19
  4.1 Algorithm Overview ............................................................. 19
  4.2 Server Responsibilities .......................................................... 20
  4.3 Agent Responsibilities ........................................................... 21
  4.4 Preliminary Algorithm Setup .................................................. 22
    4.4.1 The $\mathbf{R}$ Matrix .......................................................... 22
    4.4.2 The $\mathbf{P}$ Matrix .......................................................... 23
    4.4.3 The $\mathbf{A}$ Matrix .......................................................... 23
    4.4.4 The $\mathbf{E}$ Matrix .......................................................... 24
    4.4.5 The $\mathbf{s}$ Vector ............................................................ 24
  4.5 The Calculations ..................................................................... 25

5 IN-DEPTH DISCUSSION OF THE ALGORITHM ........................................ 26
  5.1 Explain Decisions Made ........................................................ 26
    5.1.1 Resources ....................................................................... 26
    5.1.2 Trust .............................................................................. 28
    5.1.3 Significance of the Values ............................................... 31
  5.2 Various Uses of the Algorithm ............................................... 34
  5.3 Strengths and Weaknesses of the Algorithm ......................... 36
    5.3.1 Strengths ....................................................................... 36
    5.3.2 Weaknesses .................................................................... 39

6 EXPERIMENTATION ........................................................................... 43
  6.1 Boundary Cases ....................................................................... 45
    6.1.1 Test #1: Only One Server Has All the Necessary Resources ...... 45
    6.1.2 Test #2: One Server Has Much Better Resources Availability ...... 47
    6.1.3 Test #3: One Server Has Much Better Resource Evaluations ...... 49
    6.1.4 Test #4: One Server Permits Much Greater Quantity of Resources ... 51
6.1.5 Test #5: Best Resources, but Lowest Evaluations
6.1.6 Test #6: Perfect Evaluations, Much Fewer Resources
6.1.7 Review of Results
6.2 Print Job Scheduling Simulations
6.2.1 Simulation #1: Only One Printer Has All the Necessary Features
6.2.2 Simulation #2: Same Job to Similar Printers
6.2.3 Simulation #3: Same Job to Printers with Changing Evaluations
6.2.4 Simulation #4: Different Printers with Changing Job Specifications
6.2.5 Review of Results

7 FUTURE WORK
7.1 Specifying a Server Preference
7.2 Different Weights for Different Evaluations
7.3 Different Types of Evaluations
7.4 Evaluation Algorithms
7.5 Handling Dynamic Resource Availability
7.6 Non-Linear Resource Ranking
7.7 Universal Naming Scheme
7.8 Resource List from Task Description
7.9 Deciding on Risk Factor and Resource Importance
7.10 Server Rules and Advertising
7.11 Server and Agent Police
7.12 Testing the Algorithm in Specific Environments

8 CONCLUSION

APPENDIX A: SOURCE CODE
Appendix A.1: Algorithm Tests
Appendix A.2: Printer Simulations
Appendix A.3: Support Files

APPENDIX B: DATA FILES
List of Figures

Figure #1: Simulation of one usable printer .................................................................62
Figure #2: Simulation of submitting the same job to similar printers .....................64
Figure #3: Simulation of changing printer evaluations with 30% risk .....................66
Figure #4: Simulation of changing printer evaluations with 80% risk .....................66
Figure #5: Simulation of changing job feature importance factors at 10% risk ..........68
Figure #6: Simulation of changing job feature importance factors at 20% risk ..........69
Figure #7: Simulation of changing job feature importance factors at 90% risk ..........69
List of Tables

Table #1: Review of all test results ..................................................................................58
Table #2: Algorithm testing data file locations .................................................................134
Table #3: Printer simulation data file locations .................................................................134
Glossary

**Agent**: page 3
a self-contained software entity capable of carrying out a set of tasks, requested by a human or another software entity, either locally (on the computer where it was instantiated) or remotely (on a computer other than where it was instantiated).

**ACL**: page 2
Agent Communication Language.

**AES**: page 4
Agent Environment Server. A server that is capable of providing the necessary resources to allow an agent to execute on a computer system. To agents, an AES is analogous to a fish tank for fish.

**Agent Author**: page 7
The person or group who takes responsibility for the creation of a particular type of agent.

**ATM**: page 26
Asynchronous Transfer Mode. A high-speed computer network technology.

**ATP**: page 5
Agent Transfer Protocol.

**CPU**: page 10
Central Processing Unit. More commonly used to refer to the microprocessor in a personal workstation computer.

**FTP**: page 5
File Transfer Protocol. An application layer network protocol for moving files around a network.

**Foreign Server**: page 12
Any AES other than an agent’s home server.

**Home Server**: page 12
The AES in which an agent is instantiated.

**HTTP**: page 5
HyperText Transfer Protocol. An application layer network protocol for accessing information on the WWW.

**JVM**: page 6
Java Virtual Machine.
KQML: page 2
Knowledge Query Management Language. An agent communication language (ACL).

MA: page 6
Multiple Agent. Often used in a sentence as “MA system”.

RAM: page 15
Random Access Memory. Volatile memory inside a computer.

SMTP: page 5
Simple Mail Transfer Protocol. An application layer network protocol for sending and receiving electronic mail.

TCP: page 5

User: page 1
The human with the most direct causal relationship to an agent’s instantiation.

WWW: page 26
1 Introduction

1.1 Agent Description

In the modern world, computers are rapidly being improved with the discovery of new technologies, the rediscovery of “old” technologies, and the merging of several different technologies. Software agents are a prime example of the collaboration of research efforts involving new and old technologies being combined and transformed into new technologies with great potential for leading technological advances into the new millennium. [1] describes agents as “the next great wave of innovation and development across the Infosphere” and which will “have an effect as profound as the World Wide Web” on humans all over the world.

Currently there is no globally accepted definition of software agents (referred to as “agents” from here on), but there are many agreed upon characteristics that are common among many different groups of agent researchers. The core concept of agents is that they are computer code that has a state associated with them in much the same way that “objects” do in Object Oriented Programming (OOP). Beyond this basic core, there are other characteristics that are dependent on each researcher working in the field of agents. Here is a list of some of the common agent characteristics.

- **Autonomous**: According to [1,2,3,4,5,6] agents should be able to act independently of the rest of the computer system. In operating system lingo, an agent would be similar to a “process”. An agent can perform tasks without direct control from some external source and is a software entity that has very definite boundaries.

- **Emissary**: [2,3,5,6,7,8] mention that an agent performs tasks on behalf of human users or other agents. When an agent performs a task on a computer system, it is doing so in place of a particular human or another agent. Anything that an agent does could have been done by a human and is, in fact, done for that human. If an agent uses electronic currency to purchase something, the human, who sent the agent, actually purchased the item.
Reactive (Robust in changing environments): [3,4,5,6,7,8] agree that computer systems are generally dynamic systems, so an agent must be able to cope with such an environment. When the environment changes in such a way that it affects an agent, the agent should be able to respond to the change. If an agent is printing documents for a user, and a new printer is added to the system, the agent should be able to decide to use the new printer if it is appropriate. Agents should also be able to handle similar scenarios that are negative in nature for the agents.

Intelligent: [1,5,6,7,8] claim that agents should have a minimal ability to reason about their situations and apply previously accumulated knowledge to make decisions in their dynamic environments. The greatest difference between intelligence and being reactive is that a purely reactive agent will always respond to the same environment in exactly the same way. With intelligence, an agent may respond differently at different times to the same environmental situation because it may have acquired knowledge since the previous time(s) that it was in the same situation.

Communicative (with other agents and/or human users): The idea of communication for agents is mentioned by [2,3,4,5,7,8,9] as they discuss how agents can accomplish certain tasks. [7] specifically focuses on the idea of creating Agent Communication Languages (ACL) that would “let heterogeneous agents communicate” with each other. One such ACL is the Knowledge Query Management Language (KQML); KQML allows agents to “tell facts, ask queries, subscribe to services, or find other agents.” Communicating with humans can be as simple as opening a window on the user's workstation and displaying a text message in it as is demonstrated in [2]. However, messages among agents require a specific protocol and some sort of message passing infrastructure within the agent system.

Collaborative (with other agents and/or human users): [3,4,5,7,8] refer to agent systems in which an agent will work with other agents or a human user in order to accomplish the task that the agent is trying to perform. [7] specifically deals with agents collaborating with other agents, whereas [8] describes an agent as a
“personal assistant who is collaborating with the user in the same work environment”. Agents can also be used to “help different users collaborate”.

- **Drudge**: One of the most basic uses of agents has been to establish agents as software that can take some of the boredom out of people's lives by allowing computers to perform more of the tedious and repetitive tasks that people dislike doing. [2,5,6,8] have addressed these issues in various forms. According to [8], an “agent can assist in … information filtering, information retrieval, mail management, meeting scheduling, selection of books, movies, music, and so forth.” “The biggest advantage that agents bring is simply their ability to automate previously manual operations.” [6] Having an agent perform these sorts of tasks will increase a person's productivity and give the person more time to work on tasks that are still too difficult for a computer to perform. [6] refers to customers who “will employ software agents to help them identify, locate, and procure the products and services that they require”.

- **Pro-active (has goals)**: Many agent researchers, such as [3,4,6,7], describe agents as having “goals” or “desires”. In particular, [7] refers to agents as having “intentions” and “social commitments”. Such goals, intentions, and so on allow an agent to perform tasks without a user specifying how an agent should act in all circumstances. So, instead of a user having to tell an agent to how to swap two values by telling it every step, a user might be able to describe to the agent what the result will be, and the agent will figure out the steps. This is closely tied to intelligence, but it differs because it is more than just reasoning; being pro-active means that an agent will reason with an “abstract” (as abstract as currently feasible using modern technology) purpose.

For the purposes of this thesis, software agents will be described as autonomous software entities that perform tasks on behalf of other agents or humans, and that have some degree of intelligence. The other characteristics that were described above seem to be characteristics of particular implementations and not characteristics of generalized software agents.
1.2 Mobile vs. Non-Mobile Agents

There are two major types of software agents:

- **Mobile**: “Mobile agents roam the network visiting various servers in carrying out their tasks.”[2]
- **Non-Mobile** (or Static): “Static agents are associated with a particular client or server”[2] and have no ability to roam around a network.

Non-mobile agents are typically “personal digital assistants” like those described in [8] or like the Microsoft Office Assistant that helps users with creating their documents. Being non-mobile, an agent doesn't have to be concerned with many problems that arise when dealing with networks. Non-mobile agents are usually not concerned with accumulating extra “baggage” during their existence because they don't need to transport that “baggage” across a network. There are few security concerns because the agent isn't migrating over “unknown” networks to possibly “unknown” environments, etc. The largest difficulty for non-mobile agents is being able to access remote resources. In order for a non-mobile agent to access a remote resource, there must be a specific remote access method available for that resource. This could become difficult as new remote resources become available with new remote access method protocols.

“If the agent can be serialized (prepared for transfer over a network in a way that lets its state be recovered) and if it migrates, it is a mobile agent.”[2] What this means is that “an executing mobile agent is a continuously executing program, interrupted briefly during transport between a series of machines.”[3]

“Mobile agents do not transport themselves”[3], instead mobile agents rely on agent environment servers (AES) to transport the mobile agents across a network. An AES provides an environment, in which an agent can execute, that is very similar to that of water for a fish. Just as a fish cannot live outside of water, an agent can only “live” within an AES. An AES provides mobile agents with a means of migrating from one server to another, access to resources, and a “guard against mobile agents which attempt misuse”[3] of the AES.

Mobile agents have some significant advantages and disadvantages over non-mobile agents. Many of the advantages are similar to the desired goals of researchers in
the area of process migration. Some of these goals are reduction of network traffic, load balancing, fault resilience, asynchronous interaction, etc. Some of the most important disadvantages or difficulties with mobile agents are multiple platform execution, network security, security of the agent on a remote server, security of a server from remote agents, human responsibility for agents’ actions, network traffic and reliability, resource discovery, resource usage, and resource control, etc.

This thesis is primarily concerned with mobile software agents. From now on, “agent” will refer to mobile agents unless otherwise specified.

Mobile agents are an excellent idea in theory, but in order for them to fulfill their potential, certain questions need to be answered to fully realize the theory of agents.

- How can an agent have its binary image and its current state transferred from one computer to another? Before an agent can migrate, “execution must be stopped and all local-resource-dependent activities have to be completed.”[10] Once this is complete, most systems use “application protocols on top of TCP (Transmission Control Protocol) for transport of agent code and states.”[10] Some systems use Simple Mail Transfer Protocol (SMTP), Hypertext Transfer Protocol (HTTP), File Transfer Protocol (FTP), etc. IBM developed its own Agent Transfer Protocol (ATP).[11]
- How can a piece of software be transferred from one computer to another in the middle of its execution? The typical response to this question is to create agents that are event driven state machines that can only migrate when they have released all of the local resources and their execution is currently at the event handling procedure.[10]
- How can an agent migrate among a heterogeneous set of computer architectures? Most designers are using some form of interpreted language. One of the favorite interpreted languages is Java byte code.[10]
- How does an agent know where it should migrate to? Most current agent systems require that a user specify which computers the agent should migrate to and in what order.[2] In order to intelligently select a server, agents would need to perform remote resource discovery. According to [10], “resource discovery is virtually absent in all current systems.”[12] investigated two methods “for
acquiring information regarding remote hosts” in order to decide if a remote host was acceptable for a new agent.

- How does an agent decide whether a particular server is a safe place for the agent to migrate to? At the current time, there has not been much research in this area because most systems require the user to predetermine the agent's agenda (or itinerary).[2] In most MA (multiple agent) systems, the AES is “assumed to be trustworthy” and will perform operations to help an “agent to protect the privacy” of its contents and results.[10]

- How can an agent carry sensitive information (such as electronic money) in such a way that during migration, or while residing on a remote server, the information won't be stolen from the agent (or modified)？ This can be accomplished with modern encryption techniques.[10] The major problem with this is that most encryption algorithms are very processor intensive and are slow.

- How does a server determine which agents are safe and should be allowed to transfer to itself? Most current implementations of agent servers don’t really handle this at all, and there isn't much need to because the servers provide agent access to very few resources.

- How does a server determine which resources can be provided to particular agents? Sun Microsystems has partially answered this question for mobile Java objects by providing programmers with the ability to implement different Resource Managers for each Java Virtual Machine (JVM) that they wish to use.[15] The Resource Manager is capable of course-grained control over the resources that an object is allowed access to.

- How can agents work together in groups to accomplish tasks? There are many researchers currently working in this area. In particular, [12] analyzed the idea of cloning agents in order to distribute tasks.

- How can a user specify tasks for an agent to perform without having to write programs in languages such as C++ or Java? There are some researchers investigating visual methods for specifying high-level tasks.[3]
1.3 Trust and Security in Multiple Agent Systems

The ideas of trust and security in MA systems are very closely related. [14] refers to trust as being a “particular level” of “subjective probability”. Trust is unique to an individual and it has various levels. Security, however, is not subjective. Security does have various levels, but those levels can be clearly defined.

Trust is a matter of confidence in someone’s or something’s truthfulness, accuracy, ability, strength or character. In terms of mobile agent systems, there must be trust between agents and agent servers. Agents need to trust AES’s to be truthful about the information that they provide to agents, and agents need to have trust in an AES’s ability to provide particular resources to the agents. Agents also need to trust other agents. If multiple agents are going to work together on a task, they all need to trust one another to a certain degree.

The first item involved with trust is the identification of agents and servers. In order for an agent to trust a server, the server must be identified as a server with which the agent is familiar. [10] refers to some agent systems that use globally unique identifiers for agents and servers. In many agent systems, when an agent requests access to a new server, the server performs a “verification of the agent”[9] to make sure the agent is trustworthy. The verification process may involve looking up the name of the agent in a database to see what level of confidence the server should have in the agent.

There are security issues involved with the identification of agents and servers. There has to be a method of identifying an agent or server in such a way that the agent or server cannot be an impostor. The systems described in [3] use agents that are “digitally signed by one or more parties using one of a number of algorithms, such as a public key signature algorithm.” “Digital signatures can be used to verify the identity of the mobile agent's author and of its sender, where and when it was sent, and that it has not been tampered with in transit”. [3]

Once the identity of an agent or server is successfully verified, there is another problem. “Authentication credentials do not guarantee that the mobile agent will be harmless, or even useful.”[3] Just because the identity of an agent is safe, there is no guarantee that the “digitally signed code has been authored by competent programmers.”[14] This problem leads us to the conclusion that “unless the mobile agent
has been tested on the present computer, there is no way of knowing if its software will cause harm to its new environment.”[3]

There are two major concepts that are involved with trust. The first item that trust is concerned with is whether or not an agent or server is going to do what it is supposed to do. If a server is supposed to permit the use of five semaphores, but it only permits an agent to use three, then the server isn’t acting in a trustful manner. The second concern is that an agent and server aren’t going to act maliciously. This is where security plays a major role. Trust is the amount of confidence that two entities have in one another to do as expected. Security involves possibly forceful measures to make sure that both the agents and servers act non-maliciously. Security issues have been studied in great depth and many mechanisms have been developed to create secure systems, but the current technologies are “lacking the complementary tools for managing trust effectively.”[14]

Trust can be developed in multiple ways. Some of these methods will be presented from the point of view of an agent trusting an agent server, but can be applied to the opposite situation just as well. The easiest method is that a user informs an agent as to which servers the agent should trust. Another method is that an agent asks other agents for recommendations as is described in [13]. A third method is that an agent uses a server and gains a level of trust from personal experience. The second method has an interesting difference from the others. If an agent is requesting recommendations from other agents, then the agent must have a level of trust in the other agents. That trust will cause a different value of trust to be placed in the recommended servers. This implies that a poor recommendation from a well-trusted agent could be on par with a very good recommendation from a lowly trusted agent. The other two methods of developing trust don’t involve a middle trust value.

Security in multiple agent systems has three main concerns. Agents need to have their programs and data protected from malicious servers. Servers, as well, need to be protected from malicious agents. Lastly, agents need to be protected from other agents.

The first concern can be remedied by having agents and their data encrypted and digitally signed in such a way that when an agent arrives at a new server it can verify that it was not tampered with. An agent can then be verified again when it returns to the server where it originated, but it is difficult for an agent to protect itself from a malicious
server while it is executing on the server. A very realistic problem is that an agent can be easily terminated by a server, and there is very little that an agent can do about it.

The other two concerns are primarily focused on mechanisms built into the AES. An AES can protect agents from one another by placing each of them in separate “sandboxes” as is done in Java Virtual Machines (JVM)[15]. This is the preferred method of almost all multiple agent systems. An AES can protect itself by limiting an agent’s access to system resources. Most systems have something like a “reference monitor”[3] or a “security manager”[15] that provides this service. Another method of protecting the AES and other agents is to perform code verification before it is allowed to execute on the AES. An agent’s code can be checked to make sure it is free from illegal instructions.

1.4 Resource Allocation and Usage in Multiple Agent Systems

There are two main concerns for agents regarding resources. An Agent wants to know what the rules for resource access are on a server before the agent migrates to the server. Once an agent arrives at a new server, the agent is concerned with the methods for accessing particular resources.

Most current agent systems handle the situation of multiple servers providing different resources or they don’t specifically mention how they handle this situation. In systems such as that described in [2], the servers perform resource location for agents and will create more agents if it is possible to divide a task and send agents to different servers. Most mobile agent systems provide very limited access to system resources, so there is no need to advertise the resources. Especially with systems like [11], in which the agents are derived from Java applets, the JVM provides access to a well-known standard set of resources. In more complicated systems, it is foreseen that servers will provide services for many types of agents and each type of agent will receive access to a different set and quantity of resources. In these types of systems, it will be very important for agents to be able to find out what the rules are for them on different servers. By knowing the rules, agents can better decide where to migrate.

The concept of providing and limiting resource access in an MA system is referred to as “resource management.” According to [10], resource management is not
very clearly discussed in the documentation for most modern agent systems. There are many types of computer resources that need to be managed: CPU cycles, disk space, memory, network access, process handles, database engines, printers, etc. Different types of resources will require different methods of access. CPU cycles should not require any specific method other than the protocols to migrate to a particular server, but perhaps there could be a method of requesting more CPU cycles. Disk space requires opening files in different areas of the file system. There is more management required in this situation because different agents will be allowed access to different regions of the file system and they will be allowed varying amounts of space on the disks. Resource management consists of many different tasks and can thus be very CPU intensive. This may be the most important reason why it is not very well handled in most systems, but as computers become more powerful, this problem will hopefully become less of an issue.

1.5 Conclusion of Introduction

Since agents are a new technology, there are still many questions that are unanswered, and there are still many uses of agents yet to be discovered. Multiple agent systems have to deal with the concerns of trust, security, and resource management. [14] discusses many traps and pitfalls that researchers and designers should be wary of as agents grow out of their infancy. One of the greatest problems is one that has been seen repeatedly in the computer industry and it is described as: “if the only tool you possess is a hammer, then everything looks like a nail.” This implies that “there is a danger of believing that agents are the right solution to every problem.”
2 Description of the Problem

2.1 The Focus of the Thesis

Currently, there are not many mobile agent systems that have agents using much intelligence to decide on servers to which they will migrate. There are some agent systems that require the human user to specify the exact servers to which the agent will go. Other systems allow a user to specify the particular resources that the agent needs to acquire and the originating server will send the agent to the appropriate server(s) or at least create a specific itinerary for the agent.

The purpose of this thesis is to develop an algorithm that will add intelligence to many of these agent systems. There are concerns with many of these simple selection processes. Resources are servers usually have varying availability depending on the other processes (or agents) that use the servers. Most modern operating systems limit the access of resources to specific users or groups; it is expected to have the same sort of security (or access limitations) imposed on agents. Not all resources have the same importance to the completion of a task as other resources. Most current systems just assume that all needed resources are as important as each other. They also assume that if a server has the needed resources at all, then it has a sufficient quantity of each resource. Most systems also assume that there is only one set of resources that any particular agent will be interested in; those resources are absolutely necessary to perform the agent's task(s). Many agent systems don't concern themselves with issues related to trust between servers and agents.

Many of the assumptions that the other agent systems make are not considered reasonable any more. This thesis attempts to propose an algorithm that agents can use and that provides a solution to many of the assumptions mentioned above. The algorithm being proposed is meant to provide a basis upon which further enhancements can be easily made. The algorithm provides a means of taking into account the fact that resource availability varies over time among servers. When a server limits the access of certain resources to particular agents, the algorithm will use this information in the decision process. If an agent is capable of rating the importance of the various resources that it
will use, the algorithm will consider this information. Not all agents only require the same amount of each resource. An agent, using this algorithm, will be able to specify minimum quantities of each resource in which it is interested. In addition to this specification, agents can provide a listing of resources that can be necessary for the task, or may just be desired in order to act more efficiently (or any other reason). These are the issues to which the proposed algorithm in this thesis will attempt to respond.

2.2 Assumptions

Before the discussion of the solution can be discussed, this thesis requires certain assumptions. It is hoped that this solution requires less limiting assumptions than previous agent system solutions. This is the only way that this solution will help advance agent systems.

Agents are assumed to require an agent environment server (AES) in which to be active. An agent cannot execute its code outside of an AES. The terms “AES” and “server” will be used interchangeably throughout this thesis. Agents must be created on a “home” server. Any other AES is considered to be a “foreign” server. An Agent is expected to always return to its home server after it has completed its given tasks. It is assumed that agents will not replicate themselves on foreign servers because that would complicate the concept of the “home” server.

An AES can be compared with a fish tank. There is no theoretical limit as to how many AES may exist on a specific computer system. Each AES provides a separate environment in which agents can exist and just because multiple AES can reside on the same computer system doesn’t mean that they should be considered any more connected than any other set of AES.

As agents migrate around a network there are certain items that an agent is expected to carry with it. Agents need to be identified by several factors including the author of the agent, the user on whose behalf the agent is acting, the type of agent, the home server. Agents should also carry with them a list of previous servers to which the agent visited since it was created. Each of these items should be encrypted in the agent so that it cannot be modified by other agents or servers. These items will help an AES determine the amount of trust that it should have for a particular agent.
Assuming that agents can get access to a list of active AES’s, agents must ask each AES for permission to migrate to the server before an agent is allowed on it. A permission request should include items that identify the agent, as described in the previous paragraph. The agent should probably also inform the server as to the agent’s current size, including its code and its data.

System administrators will be responsible for determining the rules that agents must follow on particular servers. It is assumed that an administrator can specify different levels of rules. There can be “global” rules that would apply to all agents that use a particular server. “Directed” rules can apply to specific agents or groups of agents. Some directed rules can apply to individual agents, and other rules can be applied to a group of agents as a whole. One directed rule might specify that no agents belonging to a particular group are allowed access to more than two software licenses. Another directed rule might declare that the total disk space being used by all the agents in a specific group cannot total more than ten megabytes at any time. A global rule might be that there is not allowed to be more than fifteen agents occupying the server simultaneously.

In addition to rules being posted by servers, the servers are also expected to provide agents with a means of finding out the current availability of various resources. The servers only need to provide availability information regarding resources to which the particular agent has access permission. This information should be updated at least as often as agents request the information.

Just before an agent leaves a server, the server will present the agent with a survey to fill out. The survey will allow an agent to evaluate the services provided by the server. It is expected that an agent will evaluate how well the server provided access to each of the resources that the agent used. However, agents do not have to evaluate all of the resources that they used, but an agent cannot evaluate any resources that it did not use during the course of the current visit. The reason for an agent’s evaluation of each resource is left to the agent to decide, so there will be no laws placed upon agents as to how they are supposed to evaluate the servers.

Agents are assumed to be fair and non-malicious. When an agent evaluates a server, the agent is expected to give reasonable evaluations. If an AES provides an agent with all the resources that the agent needs, it is expected that the agent will give the
server a relatively good evaluation. On the other hand, servers are assumed to not be malicious as well. An AES will respect all the agents that use its services and therefore will not lie, harm, or steal from them. Servers will also provide a reasonable amount of security for the agents. Every AES will make sure that no agent under its protection can be harmed by any other agent or process executing on the same computer system.

Although it is not a focus of this thesis, an agent is assumed to be able to communicated with other agents using an agent communication language (ACL) such as Knowledge Query Management Language (KQML). This will allow agents to work together on certain tasks (discussed in [7]).
3 Overview of the Solution

In order for agents to select a single server out of a pool of servers, it is necessary to investigate the types of information that an agent would need in order to make an educated decision. Following the description of agents, from the Introduction, an agent’s purpose is to perform tasks on computers. In order to do anything on a computer, some computer resources need to be accessed. An agent needs to be aware of the resources that it needs to perform its tasks. In other words, it is very important for an agent to know what it needs to perform its tasks before it starts to do them.

3.1 Needs and Wants

There are two major types of resources that an agent will use to perform its tasks. The first type is resources that are absolutely necessary in order to perform a task. The other type is resources that are not necessary, but would be helpful in performing the task at hand. Each of these resources is important, so both must be considered when deciding on a server to which the agent should travel.

Perhaps an agent needs at least 5MB of RAM in order to perform an operation, but if it had access to at least 15MB of disk space, the agent would be able to perform the operation much more efficiently. Then when selecting a server on which to perform this operation, both of these bits of information should be considered.

In addition to specifying whether a resource is needed or wanted, an agent would probably want to give the resource some sort of “importance factor”. This factor would allow the agent to specify numerically how important a resource is to the agent’s task. Assume that an agent needs to perform some image manipulation algorithms on some very large images. The agent might decide that it is more important to have a lot of RAM than it is to have a very fast CPU; if the computer has to keep swapping the image to the disk drive, then the fast CPU won’t help too much. In this scenario, the agent would want to consider a slightly slower CPU in order to have more accessible RAM. This should be considered when selecting a server to which an agent will migrate.
3.2 Evaluations

No matter how fast a computer is there will still be times when a computer cannot perform all of its tasks in the amount of time that a user expects or requires. In real-time systems, a processor may reach the point where no more tasks can be scheduled according to the specifications of the tasks. On a multi-user non-real-time system, it is very difficult to perform operations within an expected amount of time. A user can, however, maintain a history of when the resources were available. Using the history list, a user might be able to estimate when a good time would be to perform a task that needs particular resources.

In a multi-mobile-agent system, it is impossible to know exactly when a resource will be available on a particular computer. If some sort of history is maintained (similar to that which is described above) of times when agents were able to get access to all of the resources that they were interested in, then an agent might have a much better chance at guessing at whether or not the necessary resources will be available at a particular server.

An idea for keeping a history might be that when an agent is done performing its task on a computer system (or any time after it has finished using a particular resource), the agent can evaluate how well the server provided the particular resources to the agent. If a server provided all of the resources that it told the agent it could have, then the server would expect to get a good evaluation for that resource at that time. However, if an agent had to wait a long time to get access to a resource, the agent might make a worse evaluation. Since agents are expected to be somewhat autonomous and intelligent, it should be expected that different agents will evaluate servers according to different criteria. This evaluation policy is a lot like a democracy. If there are more agents which give a particular server good evaluations than there are agents giving the server bad evaluations, then overall the server should have a good rating.

A possible distributed implementation of this concept would be to have the evaluations maintained on the server to which they apply. This would make it very easy for an agent to be able to find the evaluations of a particular server, and it would greatly reduce the amount of baggage that an agent would carry around with itself. Of course, it would not make sense to store the evaluations on the server if the server has the ability to
modify the evaluations. There exist encryption methods, such as public and private key encryptions, which would allow an agent to encrypt its evaluations in such a way that others will be able to read them, but if anyone tries to modify them, the resulting data would not make any sense. It would probably be very useful if each evaluation was time stamped and was marked with some information which would uniquely identify the agent that left the evaluation.

This solution has an inherent problem of requiring an initial evaluation for each resource on new servers. It has been assumed that when a new server joins a mobile agent system, the system administrator will request an independent party, who is responsible for evaluating new servers, to evaluate the server. The independent party will use each of the resources on a server and leave initial evaluations. If there is a resource on a server that has not been evaluated yet, the server will return an evaluation value of 0 to the requester. This places the responsibility on the server's administrator to make sure that the server's resources are evaluated. This also answers the question of what a server is supposed to do when it adds a new resource to its list of services. Either the server administrator can have the new resource evaluated by the independent party, or the administrator will take a chance that the system will be used even with a poor evaluation for the one new resource.

3.3 Risk

Different mobile agents have different needs and goals. Sometimes an agent might be willing to take a chance and try a server that has worse evaluations than another server, but has a much greater quantity of the resources that it is interested in. The agent would be taking a chance that the server has bad evaluations from the past, but that it will do better now. This could be like going shopping on the day after Thanksgiving. In the past, it has always been one of the busiest shopping days of the year. Someone might take a chance by going shopping and hoping that many other people will stay home because of the amount of traffic in past years.

In another situation, an agent may be very conservative. The agent needs to get a task done by a certain time. In this case, the agent would probably decide to take very few chances (if possible) to get the job done by the specified time. The agent might be
willing to take a small chance with a server that has slightly lower evaluations than another, if that server has a good possibility of allowing the agent to finish the task significantly sooner than if the agent goes to the other server.

A good algorithm for allowing agents to intelligently select a server to which it will migrate should allow the agent to take some risk if it wants to or no risk at other times. The same agent might want to take different risks for different tasks that it performs. Allowing different agents to dynamically specify the risk that they are willing to take at different times provides a very flexible server selection process.
4 Description of the Algorithm

4.1 Algorithm Overview

There are three main items that an agent must consider about the servers that it is going to evaluate and choose among. Each of these items involves the resources that the server makes available to agents. The three items are:

1. **Resources Permissions:** For each resource type, the server informs the particular agent as to how much of that resource the agent is allowed to have access. The quantity of a resource that is permitted is dependent on the relationship between the particular agent and the particular server.

2. **Resource Availability:** For each resource type, the server informs the particular agent as to how much of that particular type of resource is currently not being used on the server. This value is independent of the value for the resource permissions. The server may only allow the agent 5 of some resource, but the server may currently have 200 of that resource available.

3. **Resource Evaluations:** For each resource type, the server maintains a list of evaluations that were made by past agents that used the resource. These evaluations are made available to other agents who are looking for information regarding the server’s past servicing of agents. It is assumed that when an agent is in the process of leaving a server, the agent adds its evaluations of each of the resources (that the agent used) to the current list that the server makes available to other agents. Each evaluation is on the range [0..10], where 10 is much better than 0.

The algorithm can be viewed as being composed of three major steps in which the second step is divided into two independent tasks which could be performed in parallel.

1. Find all the servers that permit at least the minimum quantity of each of the resources that are necessary for the agent to perform its task. In addition to the necessary resources, there may be other resources that the agent might be able to use in order to perform its task more efficiently, quickly, etc. These other resources will be investigated as well. The agent will need to gather information
from all of the servers on each of the resources that the agent is interested in. Only servers with all of the necessary resources should be considered.

2. Assign scores to the servers:
   (a) Rank the servers based on the importance of the resources that they have available to the agent. The agent is responsible for specifying which resources are *Necessary* and which resources are *Wanted*. Then, each of the resources must be given an importance rating by the agent.
   (b) Rank the servers based on their evaluations. The agent must specify how many (a maximum number) of the most recent evaluations it is interested in examining for each resource on each server.

3. Merge the results from the step 2 into a final ranking of the servers. The key to this merging process is a *Risk Factor* that the agent must specify. This factor represents how much risk the agent is willing to accept in order to find a server with better resources. If an agent is willing to take a greater risk, then it is willing to accept a server with much better resources than another server that has better evaluations.

**4.2 Server Responsibilities**

Each server must provide a certain amount of information in order for an agent to make an “educated” decision. Each server must provide the following information upon request from an agent:

- A listing of all the types of resources to which the server will permit at least minimal access for the particular agent making the request.
- For each resource that the server makes available to the agent, the server must specify the maximum quantity of that resource that the agent will be permitted to use on that server.
- For each resource that the server makes available to the agent, the server must provide the agent with a relatively recent report as to the current availability of the resources. This is done by specifying a count of the resources, of that type, which are currently not being used by any other agent or processes on that server.
The last piece of information that the server should provide to the agent is a list of the most recent evaluations of each of the resources that the server makes available to the agent. If no agents have used that resource on the server yet, then there will be no evaluations of it. In this case, the server has to say that it has an evaluation score of zero for that resource type. All evaluations that are available on a server must be on the range [0..10]. A larger value for an evaluation signifies a better (more positive) evaluation.

4.3 Agent Responsibilities

In order for an agent to use this algorithm, it must specify a few items to help direct the algorithm to find the server best suited to the needs of the agent. These particular items are:

- The agent must provide a list of all the resources that it could use to perform its task.
- For each resource that it specifies, the agent must note whether the resource is "Necessary" for the task, or just "Wanted". The algorithm will only choose among the servers that have all of the "Necessary" resources.
- For each resource in the list, the agent must specify the minimum quantity that the agent must have available to it in order for the resource to be useful in performing the task. For example, an agent might "need" 5MB of disk space to perform its task, so a server that only permits the use of 2MB of disk space is not a possible option.
- The last item specified for each resource is the importance of the resource to the agent's task. This must be a value on the range [0..10]. A larger value signifies a greater importance. For example, an agent's task might have a greater need for more disk space than more CPU time, so the agent might rate the importance of disk space with a 9, but CPU time might be rated 4. They can both be specified as "Necessary", with minimum amounts, but this allows the agent to specify which resources are more important, to the agent, than others.
- The agent must specify a "Risk Factor" that it is willing to accept. This factor must be on the range [0..100] and represents a percentage. If the agent is not
willing to accept any risk (a value of 0%), then the algorithm will select the server with the absolute best evaluations (as long as the server has the necessary resources). However, if the agent is willing to accept 100% risk, then the algorithm will select the server with the best set of resources (no matter what the evaluations are).

- Lastly, the agent must specify a maximum number of evaluations that it wants the algorithm to consider when rating the servers. The algorithm will always examine the most recent evaluations. The value specified must be a positive integer.

4.4 Preliminary Algorithm Setup

There are multiple matrices and vectors that must be set up before the algorithm can be executed. Each of these matrices and vectors is described below.

4.4.1 The R Matrix

The R matrix will hold the information regarding how each of the resources was rated by the agent. This matrix can be referred to as the resource matrix. The resource matrix will have the form of a square identity matrix in which each row (or column) represents a different resource. So, if there are N resources that the agent is interested in, then the R matrix is of size NxN. Instead of 1's being on the diagonal of the matrix, each of those 1's from the identity matrix will be replaced by a value representing the importance of each resource to the agent.

Assume that each of the ratings that the agent supplied for each resource was placed in vector \( \bar{r} \). Then, another vector \( \bar{a} \) is created of the same size as \( \bar{r} \), however \( \bar{a} \) will be used to adjust \( \bar{r} \) so that the “Necessary” and the “Wanted” resources can be distinguished. Each \( \bar{r}_i \) that is “Necessary” will have an \( \bar{a}_i = 10 \). Each \( \bar{r}_i \) that is “Wanted” will have \( \bar{a}_i = 0 \). Then each \( \bar{R}_{i,j} \) will be filled with \( \bar{r}_i + \bar{a}_j \). This will result in each \( \bar{R}_{i,j} \) being on the range \([0..20]\) and the rest of the matrix will be 0's. An example R matrix, following the summation of \( \bar{r} \) and \( \bar{a} \), might look like:
4.4.2 The P Matrix

The P matrix, a.k.a. the permissions matrix, is assembled by the agent in order to organize all of the information regarding the quantities of the resources to which each server will permit the agent to have access. P must have the same row ordering as R. Therefore, each row in P represents a different resource that the agent has an interest in. Each column represents a different server. So, $P_{ij}$ represents the quantity of resource to which server $j$ will permit the agent to have access. An example of $P$ with 8 resources and 5 servers would be:

\[
\begin{bmatrix}
9 & 10 & 19 \\
7 & 10 & 17 \\
4 & 10 & 14 \\
6 & 0 & 6 \\
8 & 0 & 8 \\
1 & 10 & 11 \\
9 & 0 & 9 \\
5 & 10 & 15 \\
\end{bmatrix}
= \begin{bmatrix}
19 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 17 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 14 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 6 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 8 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 11 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 9 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 15 & 0 \\
\end{bmatrix}
\Rightarrow
\]

4.4.3 The A Matrix

A is the availability matrix. This matrix is very similar to P and must have the same row and column order as P. The difference is that instead of being a collection of the permitted quantities of each resource, A is a collection of the current availabilities of each of the resources at each of the servers. An example of A with 8 resources and 5 servers would be:

\[
\begin{bmatrix}
5 & 5 & 7 & 8 & 2 \\
6 & 17 & 3 & 7 & 9 \\
7 & 1 & 14 & 2 & 8 \\
2 & 1 & 3 & 6 & 9 \\
9 & 2 & 8 & 4 & 8 \\
1 & 7 & 8 & 3 & 1 \\
8 & 0 & 3 & 2 & 9 \\
6 & 6 & 6 & 6 & 6 \\
\end{bmatrix}
\]
The last matrix to be defined is **E**. **E** is the *evaluations* matrix. Once again, this matrix is very similar to both **P** and **A**, and must be in the same row and column orders as the other matrices. The purpose of this matrix is to organize the information regarding the evaluations of each resource for each server. Each server may have multiple evaluations for each resource, so a very simple algorithm has been chosen to produce a single value: the mathematical mean. The agent specified that it wants the algorithm to consider up to \( N \) evaluations for each resource, so the algorithm finds the mean of up to the most recent \( N \) evaluations for each resource for each server. These evaluations are then placed in **E**, an example of which is:

\[
\begin{bmatrix}
27 & 25 & 47 & 18 & 62 \\
18 & 37 & 33 & 17 & 59 \\
30 & 9 & 54 & 32 & 48 \\
25 & 22 & 27 & 16 & 39 \\
22 & 29 & 18 & 14 & 48 \\
23 & 9 & 28 & 23 & 41 \\
28 & 10 & 33 & 12 & 59 \\
22 & 60 & 26 & 16 & 36 \\
\end{bmatrix}
\]

4.4.5 The **ė** Vector

The **ė** vector is a simple row vector of all 1's with a size equal to the number of resources that the agent is interested in. The usefulness of this vector is that it can pre-
multiply a matrix and produce a row vector containing the sums of the columns of the matrix. A sample $\bar{s}$ for 8 resources would be:

$$[1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1]$$

### 4.5 The Calculations

The following six instructions make up the algorithm. The algorithm begins with lots of data from both the agent and a list of servers, and in the end, the algorithm will specify a server to which it recommends the agent travel for its next task.

1. The first step is to create each of the above described matrices and vectors using only the servers that permit the agent to have access to at least the minimum quantity of all of the "Necessary" resources.
2. The first calculation produces a row vector with the same number of elements as there are servers being evaluated. This $\bar{b}$ vector will hold the scores for each of the servers in terms of their ability to provide resources to the agent.

$$\bar{b} = \bar{s} \ast (\mathbf{R} \ast (\mathbf{P} + \mathbf{A}))$$

3. The next calculation produces another vector of the same size as $\bar{b}$. This vector, $\bar{c}$, will hold the scores for each of the servers in terms of the evaluations of them.

$$\bar{c} = \bar{s} \ast (\mathbf{R} \ast \mathbf{E})$$

4. Find the maximum element in $\bar{c}$, $c_{\text{max}}$ such that $\forall_i c_i \leq c_{\text{max}}$.
5. Find all $c_i$ such that $c_i \geq c_{\text{max}} \ast (1 - f / 100)$ where $f$ is the risk factor that was specified by the agent. Place each of those, $c_i$, in the set $D$.
6. Of the servers represented by the values in $D$, the server with the highest score in $\bar{b}$ is the chosen server.
5 In-Depth Discussion of Algorithm

The algorithm that has been proposed is not the final solution to the problem. There will need to be further solutions that will provide more useful utilities to an agent. However, this algorithm will provide some insight and a base from which new algorithms can be developed. It has been suspected that the thoughts behind the algorithm will be more important than the algorithm itself, so completeness will be very important in the following sections.

5.1 Explain Decisions Made

The decisions that were made during the development of the algorithm have their foundations in distributed (or simply networked) computing. When someone decides to connect to a remote computer system, he/she usually does so because the remote computer has a resource that his/her local computer lacks. As he/she connects to a remote computer, the computer performs a security check on his/her access (perhaps a user name and password) in order to assess the trust between the remote computer and the user (and/or the user’s computer). Resources and trust become the main foci of the algorithm.

5.1.1 Resources

There are many different types of resources that are available on a computer system. Many of them can be counted, such as the amount of RAM, disk space, printers, current users, available software licenses, etc. Other resources might be singular in count but are very specific items that an agent might be looking for, such as a particular CPU type, a specific database engine, etc. Some resources tend to be more important to a task than other resources. A user might be setting up a WWW server on a computer, so the user might find that disk space is much more important than having a super fast CPU. Of the resources that could be used to perform a task, some are necessary and others are only wanted (or desired). If a user needs to download a document from another computer, a network connection to the other computer is necessary, but a fast ATM connection would only be a desired resource.
From these two ideas, the R matrix was developed. The R matrix would be organized in a way that would allow the algorithm to combine the ideas of resource importance and whether a resource is needed or wanted. In order to put both sets of resources (needed and wanted) into the same matrix, there needs to be something done to the values in order to differentiate between the two. The actual values that go into the matrix have to be different than just the importance value that the agent gives to each resource. It was decided that two slightly overlapping ranges of values would provide a nice balance between not changing the values too much, and wanting to be able to distinguish between the two sets. A range of [0..10] was chosen for the "wanted" resources, and a range of [10..20] was selected for the "needed" resources. This causes a needed resource with an importance value of 0 to have the same value in R as a wanted resource of the highest importance. Later a discussion will follow regarding how the total range of [0..20] affects the final decision made by the algorithm.

The form of the other matrices determined the organization of the R matrix. The process began by placing the values in a row vector. By multiplying the row vector by a column vector, a single value was produced representing the score of a server that can be easily compared with other servers. The significance of the values in the column vector needed to be decided upon.

The values in the row vector correspond to different resources. The same ordering of resources, from left to right in the row vector, must be applied to the column vector from top to bottom. This will cause the importance value of each resource to be multiplied by some other value. It was decided to make each value in the column vector correspond to the currently available quantity of each resource. This assumes that a greater availability of a resource would be more desirable. The currently available quantity is assumed to be proportional to the quantity that will be available when the agent migrates to the server in most situations. By placing the values for the currently available quantities of each resource into a column vector, this can be multiplied by the row vector of importance values and a single value can be formed.

The equation, \( \vec{b} = \vec{s} \times (R \times A) \), can be written by giving each of the elements of the equation a different responsibility. A is meant to hold the values corresponding to the availabilities of each of the resources for each server. R maintains the importance
ratings of each of the resources as far as the agent is concerned. The purpose of \( \tilde{s} \) is to produce a summation of each of the columns of the product of \( \mathbf{R} \) and \( \mathbf{A} \). Due to this division of responsibilities, the \( \mathbf{R} \) matrix must be the diagonal matrix that is described in the previous chapter.

The resulting \( \tilde{b} \) vector will hold a score for each server in terms of the availability of each of the resources that the agent is interested in. The scores are weighted according to the importance of each resource to the agent. Each of these values can be compared with one another and sorted to see which servers have the best availability of resources.

5.1.2 Trust

There are two types of trust. Direct trust means that I have past experience that causes me to trust someone. Indirect trust involves an intermediary whom I trust and who trusts the third person. Since trust is bi-directional, the concept of trust can be described as two vectors. Each of those vectors is independent of the other. One vector may be direct and the other might be indirect. The vectors may be any combination of directness. Further, if both vectors are indirect, they can each go through different intermediaries. I may trust a bank because my parents recommended it, but the bank may trust me because the bank communicated with a credit report institution.

Indirect trust can then be divided into two categories. The first type of indirect trust involves a situation similar to one in which a person asks any number of other persons for a recommendation of yet another person. The second type of indirect trust involves a situation similar to a prospective student providing letters of recommendation to a scholarship committee. In the second situation, the student might have received the letters from previous or current teachers and the letters were placed in sealed envelopes so that the student can not tamper with them.

In an agent-server system there is trust that an agent has in a server and vice versa. The trust that a server has in an agent was the first problem that was of concern. The view was taken that this has been solved many times already and this thesis was not concerned with presenting another solution. The basic solutions have been implemented in many networked (and even some non-networked) systems. The two most common
solutions are that the agent would provide an identifier and a password, and the other
would involve an agent providing certain credentials given to it by some other trusted
server or agent. It was assumed that the servers will use some method for deciding what
resources a particular agent is allowed access to. What was of concern is that agents are
able to find out which resources and the quantities of those resources that the agents are
allowed to access. As long as an agent is able to get this information, the agent can make
a much better decision when selecting a server. It was decided that these values should
be placed in the matrix $\mathbf{P}$.

How are the values from $\mathbf{P}$ supposed to be used in the algorithm? It was decided
that $\mathbf{P}$ is very closely related to $\mathbf{A}$. $\mathbf{P}$’s contribution to the selection should have the same
weight as $\mathbf{A}$ because it was assumed that neither is more important than the other. So, it
was decided to have an $\mathbf{R} \times \mathbf{A}$ factor somewhere in the equation. Since $\mathbf{A}$ deals with
resource quantities, it made sense to put this factor in the equation for $\tilde{b}$. If $\mathbf{A}$ and $\mathbf{P}$ are
going to have the same weight then it made sense to just take the summation of $\mathbf{A}$ and $\mathbf{P}$
to form $\tilde{b} = \tilde{s} \times (\mathbf{R} \times (\mathbf{P} + \mathbf{A}))$.

The next item to deal with was the problem of the trust that an agent has in a
server. The idea of indirect trust is similar to letters of recommendation. This idea was
preferred because it means that agents don’t need to try to contact other mobile agents to
get recommendations. From past experience, it is much more difficult to contact a busy
person to ask them a question than it is to connect to the WWW and look at information
provided by a stationary server. There is also the problem of the person carrying the
information around with them everywhere they go. The same ideas hold true for an
agent. It was desired to minimize the amount of unnecessary inter-agent communication
as well as to minimize the amount of extra baggage that an agent would need to transport
everywhere. It was decided that servers could hold “letters of recommendation” from
previous agents who had used the server. These letters are referred to as evaluations.
Whenever an agent feels that it is appropriate to evaluate a server, the agent should be
able to leave an evaluation with the server. The evaluations would need to be encrypted
in such a way that the server can not modify the evaluations, but anyone should be able to
read the evaluations.
The evaluations that an agent leaves with a server would have to have some numerical value in order to use it in an equation. The agent should evaluate the service provided by each resource that the agent accessed. It is assumed that if the resource was provided to the agent in a hassle free manner, then the server should receive a good evaluation for that resource. To keep the algorithm relatively simple, it was decided to use a standard familiar scale of 0 to 10 for the evaluations. The agent should be able to request these evaluations from any server and for any resource that the agent is permitted to have access.

Again, it made sense to put the evaluations information into another matrix, $E$, that is similar to $P$ and $A$. It was also decided that the values of the evaluations should also be weighted by the importance of each of the resources. So, the factor $R \cdot E$ was chosen to be used in the algorithm. This is when it was decided that the evaluation information should be placed in a new vector, $\tilde{c} = \tilde{s} \cdot ( R \cdot E )$, and linked to the information in $\tilde{b}$ via a risk factor.

A risk is a possibility of something going wrong. My algorithm allows an agent to specify a risk quantitatively when evaluating servers. If a server has relatively good evaluations, then an agent might be able to assume that in the future that same server will continue to receive good evaluations for serving agents well. A similar statement could be made about servers with relatively poor evaluations. So, if an agent is willing to accept a relatively large risk, then the algorithm should consider servers with relatively poor evaluations. However, if an agent chooses to take very little risk, then the algorithm should only consider servers with relatively good evaluations.

Agents need to try to perform their functions as long as there is a possibility of doing so. It is assumed that there will be many different types of agents that will evaluate servers using different standards. By making the risk factor a percentage, then the servers are compared with each other instead of with a global standard. So, an agent can specify a risk factor between 0 and 100%. The algorithm will only consider servers whose $\tilde{c}$ values are within the risk factor's percentage of the server with the greatest $\tilde{c}$ value.
5.1.3 Significance of the values

It is very important to understand how the particular range for possible \( R \) values affects the overall decision making in the algorithm.

The algorithm essentially begins with the \( \bar{c} \) values since they determine which \( \bar{b} \) values will be considered in the decision (calculating the \( \bar{b} \) vector can be considered to be preparation work as far as the whole algorithm is concerned). \( \bar{c} = \bar{s} \ast ( R \ast E ) \) shows that \( \bar{c} \) is directly proportional to \( R \). Any change in the range of values for \( R \) would just change the amount that the values in \( \bar{c} \) are spread apart.

The interesting question involves the effects of \( R \) on steps 4 and 5 of the algorithm. \( R \) effects both steps in a similar way. If the range of \( R \) were increased from \([0..20]\) to \([0..100]\), then it would just involve multiplying \( R \) by 5. If \( R \) is multiplied by any constant \( k \), then \( \bar{c} \) could be rewritten as \( \bar{c} = \bar{s} \ast ( k\ast R \ast E ) \). Therefore, we could also write \( \bar{c}_i = \bar{s} \ast ( k\ast R_i \ast E_i ) \). Then, step 4 could be rewritten so that \( \bar{c}_{\text{max}} \) is found such that \( \forall_i (\bar{s} \ast ( k\ast R_i \ast E_i )) \leq \bar{s} \ast ( k\ast R \ast E_{\text{max}} ) \). If both sides of the equation are divided by \( k \), we end up with \( \forall_i (\bar{s} \ast R_i \ast E_i ) \leq \bar{s} \ast ( R \ast E_{\text{max}} ) \) which is exactly what we started with. So, changing the range of values for \( R \) would not effect step 4 of the algorithm.

The same logic can be used for step 5 of the algorithm. Changing the range of values for \( R \) is the same as multiplying \( R \) by a constant \( k \). Therefore, we could rewrite the equation in step 5 to be \( \bar{s} \ast ( k\ast R \ast E_i ) \geq \bar{s} \ast ( k\ast R \ast E_{\text{max}} ) \ast (1 - f /100) \). Again, this equation can have \( k \) divided out of both sides and we end up with the same equation that we began with. The selection of the particular range, \([0..20]\), makes sense because it allows an agent to specify values on the range \([0..10]\) which is easy for humans to deal with since we use the base 10 number system most often.

Once again, the same logic can be applied to show that the particular range of values for \( R \) does not affect the outcome of the vector \( \bar{b} \), however, it is a little more simplistic. By inspection, the equation \( \bar{b} = \bar{s} \ast ( R \ast ( P + A ) ) \) can be seen to have \( \bar{b} \propto R \). Since the algorithm inspects the individual elements of \( b \) in order to find the maximum value out of a subset of the values in \( b \), we could rewrite the last step of the
algorithm to mention $\bar{b}_{\text{max}}$ as the largest $\bar{b}_i$ from the set D. In this case, the algorithm would be comparing $\bar{b}_i$ with $\bar{b}_j$ where $\bar{b}_i=\bar{s}*(R*(P_i+A_i))$ and $\bar{b}_j=\bar{s}*(R*(P_j+A_j))$. For completeness, if we replaced R with k*R, then the comparisons would be between $\bar{s}*((k*R)*(P_i+A_i))$ and $\bar{s}*((k*R)*(P_j+A_j))$. If we divide both sides by k we end up comparing $\bar{b}_i$ with $\bar{b}_j$ again.

If R’s particular range doesn’t matter, then what role does R play in the algorithm? The easiest way to see the connection is to use a simple example of two servers each having two resources. Assume that the product $\bar{s}*R$ is $[a\ b]$ and the E matrix or the sum $P+A$ is represented by $\begin{bmatrix}w & y \\
x & z\end{bmatrix}$, then $\bar{c}$ or $\bar{b}$ can be represented by $[aw+bx\ ay+bz]$ which will be referred to as $\bar{d}$. To analyze the effects of R, we will take the limit of $\bar{d}$ as the two values of $\bar{s}*R$ are modified.

- When $a$ approaches 0, we see that the comparison between the two servers only happens between $x$ and $z$ because $b$ can be divided out.
  $$\lim_{a\to0} \bar{d} = [bx\ bz]$$

- When $b$ approaches 0, we see that the comparison between the two servers only happens between $w$ and $y$ because $a$ can be divided out.
  $$\lim_{b\to0} \bar{d} = [aw\ ay]$$

- When $a$ approaches the maximum value of 20 and $b$ approaches a very small value of 1, we see that the comparison between the two servers mostly happens between $w$ and $y$ with a very small contribution by $x$ and $z$.
  $$\lim_{a\to20,\ b\to1} \bar{d} = [20w+x\ 20y+z]$$

- When $b$ approaches the maximum value of 20 and $a$ approaches a very small value of 1, we see that the comparison between the two servers mostly happens between $x$ and $z$ with a very small contribution by $w$ and $y$.
  $$\lim_{a\to1,\ b\to20} \bar{d} = [w+20x\ y+20z]$$
It can be seen that the values of $\mathbf{R}$ can have a large effect on which servers are considered eligible for selection and it also can have a large effect on which server is selected in the end. The limits calculated above have slightly different implications depending on which vector is being calculated. When the $\mathbf{c}$ vector is being calculated, there is a strict range of values for $w$, $x$, $y$, and $z$. Each of these can only be on the range $[0..10]$. Analyzing the last limit calculation, we see that if $x$ is 10, then the closest that $y$ can possibly bring the second server’s score to the first server is within $95\% \left( \frac{(20)(10) - (1)(10)}{(20)(10)} \right) \times 100\%$. If the specified risk factor is greater than 5%, the contribution of $z$ is necessary if the second server is going to have a chance of being selected.

This situation is slightly different for the $\mathbf{b}$ vector calculation because there are no limits to the values for $w$, $x$, $y$, and $z$. The other difference is that instead of the values needing to be within a certain percentage of each other, the largest value in the $\mathbf{b}$ vector will be selected (assuming it is eligible). Using the last limit calculation again, we’ll assume that $w$ and $z$ are zero. Using these assumptions, the second server could still achieve a higher score if $y > 20 \times x$, and this could definitely be possible for certain resources.

The values in the evaluations matrix ($\mathbf{E}$) have a big effect on the final decision. As mentioned earlier, the effect of $\mathbf{E}$ on the decision is somewhat bounded by the fact that the values in $\mathbf{E}$ must be on the range $[0..10]$. The reason for bounding the values is to provide some regulation on the evaluation scores. Without this rule, one agent might assume that a very good score is 10, but another agent would consider 10 to be one of the lowest scores possible. The effects of $\mathbf{E}$ are well covered by the previous paragraphs that discussed the effects of the $\mathbf{R}$ matrix.

The values in the permissions ($\mathbf{P}$) and the availability ($\mathbf{A}$) matrices are not bounded by the algorithm. If a server happens to have one hundred printers attached to it and they are all currently available, then the server should be able to declare that it has that many printers available. Since $\mathbf{P}$ and $\mathbf{A}$ have the same weights associated with them, it is possible that two servers could end up with the same score although one server has
large \textbf{A} values, but small \textbf{P} values, and the other server has large \textbf{P} values, but small \textbf{A} values. It makes sense to give a server a good score if it permits the use of at least the amount of resources that an agent requires, and there are many of each resource currently available. However, the algorithm will also give a good score to a server that permits the use of many of each resource, but doesn’t have many available. This probably doesn’t seem fair, but it is expected that if the agent selected the server with very few resources available, then the agent will end up giving the server poor evaluations. Eventually, the server with very few available resources will not be eligible for selection due to its low evaluation scores (assuming a relatively low risk factor). The algorithm was designed to have these types of checks and balances in it.

It is very difficult to quantitatively analyze the effects of the risk factor. The values must be between zero and one hundred percent. It was decided that when anyone makes a decision, the person considers some amount of risk when making the decision. Some people naturally prefer to take more risk than others are. When someone prefers to take a higher risk than someone else does, it means that one person is willing to accept a larger chance of loss in return for a chance at a greater gain. This is exactly how the risk factor works in this algorithm. When an agent specifies a larger risk than another agent does, it means that the first agent is willing to try servers that have a better chance of not being able to service the agent. However, there is also a chance that one of those servers will be able to service the agent much better than the server to which the other agent migrates. Every different situation will cause the risk factor to have a greater or lesser effect on the decision. If all the servers in one situation have very similar evaluation scores, then the risk factor will have very little effect. In a different situation, all the servers might have drastically different evaluation scores; the risk factor will have a much greater effect in this situation.

### 5.2 Various Uses of the Algorithm

This algorithm was designed to be a good basis for other algorithms to begin with for specific situations. The initial purpose for the algorithm was for mobile agents, but since the inception of the idea, it was realized that the algorithm has other uses.
The first other use is explored more deeply in the experimentation chapter in which the algorithm is used to schedule print jobs on a cluster of printers. In this situation, the algorithm works well because it provides flexibility and intelligence that is not often found in print environments. The algorithm allows the job scheduler to take into account the past performance of printers. If a particular printer starts to jam more often than before, lower risk jobs will start to avoid that printer. Sometimes a user has different desired features for a print job and would like to specify them to the software. For example, a user might prefer his/her job to be printed on 3-holed paper. However, if the job can be printed a half-hour sooner by printing it on plain paper, the user would prefer that it be printed sooner. This algorithm easily allows for these types of situations.

The algorithm could be used for general distributed load balancing. Since the algorithm takes into account some history of servers, it would be plausible for the algorithm to be modified slightly to enhance its estimation abilities. The better that the algorithm can guess as to what the resource availability will be like in the near future for different servers, the better the algorithm will be. By packaging up processes as agents, the agents could re-evaluate their situation every so often in order to decide if they should migrate to another server.

This type of algorithm could be used for WWW searching. When a user performs a search, an agent investigates several servers with information that appears to have something in common with the user’s search parameters. When the servers’ addresses are returned to the user, the user is able to evaluate the worth of each server’s information. These evaluations can be used for future searches. If a server previously did well with providing a user with the information he/she requested, then it might have a better chance of having the requested information in the future.

Probably the most interesting future use of mobile agents is the field of electronic commerce. If an agent can shop around for the best price for something and purchase it on behalf of a human user, then the agent would definitely be interested in the particular resources (or items for sale) at different servers and would want to know how well other agents had been treated in the past. This would be an excellent way to shop because an agent could read reviews of stores by other agents that had shopped there before.
Reviews by other customers are probably much better than seeing advertisements by the company itself.

Another possible use of this algorithm is in situations involving redundant data servers like many FTP servers. Often FTP sites provide a listing of alternative sites to which people can connect. If the servers had historical information in the form of evaluations, then some entity could suggest a server to the user as being a good server for the particular location, connection speed, time of day, etc. To extend this idea, an agent could be used for retrieving data for a user. If there is a long list of items that need to be downloaded, the algorithm could provide even more intelligence to the server selection process. The agent could group the data into different sets and connect to different servers for each set depending on which servers can provide the particular bits of information as well as the evaluations from past agents.

5.3 Strengths and Weaknesses of the Algorithm

There are usually many strengths and many weaknesses of any algorithm. There are advantages and disadvantages to using different algorithms that provide a similar service. The algorithm in this thesis is no exception. In the following two sections, some of the strengths and weaknesses will be presented in order to allow this algorithm to be compared with other algorithms.

5.3.1 Strengths

One of the most important strengths of this algorithm is that it is a very generic algorithm. As long as a computer resource can have numeric values associated with it for an availability count and a permission count, then the resource can be used in this algorithm. Sometimes a little creativity is necessary to figure out how to take a resource and assign numeric values for it, but it can be done for almost any computer resource. In the experimentation chapter, the algorithm will be used as a print job scheduling algorithm. It is important to provide an ability for a user to specify a minimum amount of time that he/she is willing to wait for the job to print. This does not seem like a resource at first, but with a little thought, an equation will be developed to provide a value for a combined availability and permission matrix. There will also be an evaluation field provided so that each agent can evaluate the timeliness of its job's completion. There are
many other types of resource that may require an added equation or two, but it should be possible to fit them into this algorithm.

Allowing an agent to specify an amount of risk that it is willing to accept is a great strength of the algorithm. As machines become more intelligent, scientists tend to make the machines mimic human intelligence more. Associating a risk with a decision is a great ability of humans. Agents are meant to work on behalf of humans, so it seems that an agent should carry as much of the decision-making abilities of its user as possible. If a human were choosing a server, the human would associate some risk with the decision, so the agent should as well. The risk factor allows for the two extremes as well as a good range of values in between.

The evaluations from previous agents are another very good strength of the algorithm. Not only do they provide a check on servers, but they also allow agents to be much more intelligent about their decisions. It would be very useful if this type of evaluation concept was realized for department stores that advertise excellent sales on items of which they have very limited stock. Many people find that very aggravating, but the stores achieved their goal of getting people to come to their stores. If someone could check out evaluations from other shoppers, he/she would be able to find out that a particular store usually runs out of the sale items very quickly so it is not worth his/her time to travel to the store. This type of information makes the algorithm very useful for selecting servers.

The algorithm does not specify exactly how an agent should evaluate a server. This is another way in which the algorithm is very generic. In different environments, agents may use different criteria for the evaluations. On a particular LAN, the agents may be configured to evaluate servers based on how long it took to perform a particular task on the server. However, agents that traverse the Internet may also be concerned with the security of the data that they are carrying. There are many different ways in which agents can evaluate different servers. Agents are not forced to evaluate servers if they choose not to; an agent can even selectively evaluate different resources on a particular server. This algorithm is not rigid in any way about how the agents should rate servers.

Currently the algorithm uses the evaluations of servers in a relative manner. Since different agents will use different evaluation criteria, it is difficult to use a rigid
comparison scheme for the evaluations. A particular agent environment might lead to all the agents giving maximum evaluations of, say, seven for the best servers. If the server evaluations were not compared relative to the scores for the other servers, then it would be possible that some agents would never perform their tasks. If a minimum required evaluation score were specified, it is possible that no servers would ever satisfy the requirement. This is the strength of using relative comparisons for the evaluation scores.

Having the evaluations maintained on the servers is a good quality because it reduces the network traffic when transporting agents. If an agent had to carry around a listing of every server that it had visited and an evaluation of the server for each time that it visited, the agent could tend to become very bulky. By placing the evaluations on a server, the data only needs to be transported to agents when they are making a decision about a server. By properly encrypting the data on the servers, there shouldn’t be much concern about the servers modifying the evaluations for some malicious purpose. By maintaining all the evaluations for a particular server in one location, an agent doesn’t need to search all around a WAN in order to locate agents that have evaluated the particular server. If an agent can connect to a server to retrieve information from it, then there is a very good possibility that the network connection will be available for the agent to migrate to the server. It is also more efficient to gather the information from one location where it can be compressed into a single package, instead of having to gather many small packages from multiple locations.

The algorithm allows agents to provide a different set of required resources as well as a different risk factor each time that the agent uses the algorithm. An agent can even run the algorithm multiple times with different risk factors if it helps the agent make a decision. Perhaps an agent wants to run the algorithm for zero, ten, and twenty percent risk factors and then choose the server that is selected by the algorithm the most times (if there is one). The algorithm is not very large in terms of code-size, and the operations that it performs are very well suited for modern computer systems.

Now that some of the strengths have been analyzed, it is time to look at some of the weaknesses that the algorithm has.
5.3.2 Weaknesses

Probably the biggest weakness of the algorithm in its current form is the flexibility in specifying resources for which the agent is searching. Currently, it is not possible to list a bunch of resources of which the agent needs at least one. An agent may need to use either a tape drive or a 3.5” floppy drive. This type of specification is not currently possible. This is closely related to the problem that an agent can’t just specify that it needs any resource that fits in a general category. For example, an agent can’t specify that it needs some type of visual output device, which could include devices such as printers, monitors, LCD displays, etc. It also is not possible to specify that if a certain amount of one resource can be acquired, then another type of resource is not necessary. An agent might need to perform some storage-space intensive algorithm and if sufficient virtual memory is available, the agent doesn’t need any temporary disk space. An agent cannot specify a resource by describing the resource’s characteristics. The name of the resource that the agent specifies must match exactly the name of the resource as the server advertises it.

Another related problem with resources is the lack of flexibility in the specification of the importance factor. Some resources aren’t too important unless a certain amount can be acquired, at which point the resource becomes very important and useful. An agent may be trying to find a network over which it can deliver a message to someone. With network bandwidths up to a certain amount, the agent will deliver the message in text format. However, if the network bandwidth reaches another level, the agent will be able to deliver the message with speech. If the agent is able to find a network with sufficient bandwidth, it can deliver the message with full-motion video as well. In this scenario, an agent would want to specify the minimum bandwidth needed to deliver the text message. If the located bandwidths reach other levels, the importance of the bandwidth would need to increase. By increasing the importance factor, it conveys to the algorithm that better evaluations are needed and that extra bandwidth is better in order to maintain a fluid transmission. There are probably many other examples of resources that would have different importance values depending on the amounts that are available.

Another resource related problem was mentioned earlier in the thesis. The problem involves the availability of resources being weighted as much as the usage
permissions. This can be a problem if a server permits large amounts of resources, but doesn't have any available. That server might be selected over another server that doesn't permit as many resources, but does have a whole bunch available. This is expected to be evened out over time by the evaluations, but the first few agents will get stuck waiting for resources.

In terms of the evaluations part of the algorithm, there are many limitations there as well. The first weakness is that an agent can only specify one value representing its evaluation of a resource. It might be much better if there were different criteria upon which the agents would evaluate the resources. Perhaps one of the criteria would involve the length of time that the agent had to wait for a resource, and another might correspond to the quantity of resources that the agent was able to use. Different criteria would allow an agent to better clarify the information that it is trying to convey via the evaluation.

Another weakness in the current evaluations system is that an agent cannot specify which evaluations it considers to be important. For example, someone might want to inform his/her agent to completely trust evaluations from Barney and Betty, and only partially trust evaluations from Wilma and Fred. Currently the algorithm doesn't allow an agent to specify this type of selection and weighting of evaluations.

Two features of the algorithm that are listed as strengths can also be listed as weaknesses. Maintaining the evaluations for a server on the server itself does cause some problems. The first problem is that it is up to the server’s discretion as to how long it should retain past evaluations. Just because a server isn’t moving around a network doesn’t necessarily imply that it has a lot of extra storage in which it can keep records of agents that had once used its services. Another problem is that an agent may have to contact many servers to gather information on them, but some of them may be completely inappropriate for the agent. A possible solution is to have a few servers similar to the "yellow pages servers" described in [2] that an agent can use to quickly narrow its search by finding certain information about all the servers at one location. Once the agent has found a few likely candidates, the agent can focus more on those servers for the selection. There is another problem that is more related to security issues. If a server maintains the evaluations of itself, then it has the ability to dedicate many of its resources to decrypting the evaluations. This is not very likely to be a problem, but it
definitely is a concern, even with modern encryption schemes. The other feature that is also a weakness is the fact that the evaluations are graded relative to the other known available servers. If there is a closed network in which a specific type of agents perform their jobs, then it is possible that all the agents would use the same evaluation scheme such that they use the whole range of values. So, a good server would receive values approaching ten, and poor servers would receive values approaching zero. Then if all the available servers have evaluation scores around the value of two, the user may not want the agent to go anywhere. By evaluating the servers relatively, the agent will migrate to a server as long as one of them permits the use of the minimum quantity of each of the necessary resources.

An agent may find that it has had good luck with one particular server. In a situation like this, the agent might want to inform the algorithm that the one particular server should be given preference over any other servers. Perhaps the algorithm would weight the scores for the one server a little more than the scores for the other servers. Doing this would cause the preferred server to be selected even if its scores were just a little below those of another server. The algorithm in its current form does not allow an agent to make such specifications.

Depending on the number of servers and resources that an agent is considering, the matrices in the algorithm could become quite large. If an agent were performing the algorithm itself, these matrices would be occupying the possibly limited amount of memory that the server allows the agent to use. These large matrices may make it necessary for an agent to return to its home system before the agent can decide where to go next. Perhaps there is a way to perform the algorithm in smaller steps so that the amount of memory required at once is not so demanding. This problem may also be moot due to greatly decreasing costs of computer memory.

The last point to be discussed is the concept of collaboration among agents. It might be necessary to have two different agents meet on one server in order to perform a task. This would require that the algorithm take into account the needs of both agents when selecting a server at which they can meet. Currently the algorithm is only set up for one agent although one of the agents could find out from the other agent as to what
resources it needs. Then one agent would perform the algorithm with the combination of the two agents’ resource lists.

The algorithm is found to have a few weaknesses, but most of them can be overcome by minor modifications or additions to the current algorithm. There are probably many other weaknesses that will stand out more or less in particular situations.
6 Experimentation

The way in which this algorithm was tested was to first test several boundary cases in order to gain some credibility for the algorithm. The second set of experiments involved simulating the use of the algorithm in a realistic environment to gain further credibility for the algorithm.

This algorithm takes, as inputs, lots of information upon which it applies several calculations and sub-algorithms in order to produce a single value. For a relatively simple scenario with five servers and four resources, the algorithm would be analyzing sixty-four values (four values for the R matrix, and twenty for each of the P, A, and E matrices). These values are in four disjoint orthogonal sets. This tends to be too much information for a human to look at and comprehend without having some formal algorithm available. However, with boundary cases, a human reader could look at the data and make a decision about which server would be best suited for the agent in question.

The boundary cases were tested in six experiments where each experiment tested a different aspect of the algorithm. Brief descriptions of the purpose of each experiment are:

1. Demonstrate that the algorithm chooses the only server with all the desired resources.
2. Demonstrate that the algorithm chooses the server with the greatest availability of the desired resources, given that all the other factors are the same.
3. Demonstrate that for a relatively low-risk taking agent, the algorithm will choose the server with the best evaluations of its resources, given that the other factors are the same.
4. Demonstrate that a server, that permits a much greater use of its resources over all the other servers, will be chosen, given that all the other factors are the same.
5. This experiment is split into two different tests. The server states are the same for both tests; there is a single server that permits and has available a much greater amount of its resources than any of the other servers. However, that same server also has the lowest evaluations of its resources. The first test involves an agent
with a 100% risk factor, so the algorithm is expected to choose the above-described server. The second test involves an agent with a 0% risk factor, so the algorithm should not choose the above-described server even though it does have significantly better resource quantities.

6. The last experiment is very similar to the previous one and is also split into two tests. In this case, there is one server with perfect resource evaluations, but the quantity of resources that it permits and has available is significantly lower than that offered by the other servers. The agent with a 0% risk is expected to choose the above-described server. The agent with the 100% risk is not expected to choose the above-described server, but instead it should choose the server with the best set of resources.

The second set of experiments involved using the algorithm to simulate the scheduling of print jobs on a cluster of printers. These experiments do not demonstrate the full feature-set of the algorithm, but they do demonstrate a realistic usage of the algorithm. If the assumption is made that agents are used to transport print jobs from the user’s computer to the printer on which the job will be printed, then this algorithm could be used to allow agents to place their jobs in appropriate print queues. Each of the four experiments are described briefly below:

1. The use of a very simple printing environment is simulated in the first experiment. In a two-printer environment, only one printer has all of the needed features for all of the jobs. The expectation is that all the jobs will be scheduled on the single printer with all the needed features.

2. This experiment is only a little more complicated. This print environment assumes that there are two printers next to each other and that they are exactly the same kind of printer with the same features. Consecutive similar jobs should alternate between printers.

3. Scheduling of jobs while a printer’s evaluations begin to worsen (perhaps due to an increase in jamming, or running out of paper, etc.) is an interesting experiment. A user who specifies a relatively low risk would expect that as a printer’s
performance begins to degrade, his/her jobs would tend to not use that printer as much. This type of scenario will be tested in this experiment.

4. The last experiment demonstrates the behavior of the algorithm in the scenario of having two desired job features, which are not necessary, and two printers each having one of the desired features. The evaluations of the printers for each of those two features will be different.

**6.1 Boundary Cases**

6.1.1 Test #1: Only one server has all the necessary resources

In the following list of servers, only Server 4 permits the use of some of each of the resources that are in each table. All of the other servers don't allow the use of at least one of the resources (even though there are plenty available). This experiment verifies that with everything else the same, the algorithm selects the only server that will permit the use of all of the resources that the agent needs.

<table>
<thead>
<tr>
<th>Resource</th>
<th>Resources Permitted</th>
<th>Resources Currently Available</th>
<th>Previous Agent Evaluations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Server 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>circle</td>
<td>0</td>
<td>15</td>
<td>6</td>
</tr>
<tr>
<td>square</td>
<td>5</td>
<td>15</td>
<td>6</td>
</tr>
<tr>
<td>line</td>
<td>5</td>
<td>15</td>
<td>6</td>
</tr>
<tr>
<td>triangle</td>
<td>5</td>
<td>15</td>
<td>6</td>
</tr>
<tr>
<td>pentagon</td>
<td>5</td>
<td>15</td>
<td>6</td>
</tr>
<tr>
<td>rhombus</td>
<td>5</td>
<td>15</td>
<td>6</td>
</tr>
<tr>
<td>cube</td>
<td>5</td>
<td>15</td>
<td>6</td>
</tr>
<tr>
<td>sphere</td>
<td>5</td>
<td>15</td>
<td>6</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Server 2</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>circle</td>
<td>5</td>
<td>15</td>
<td>6</td>
</tr>
<tr>
<td>square</td>
<td>0</td>
<td>15</td>
<td>6</td>
</tr>
<tr>
<td>line</td>
<td>5</td>
<td>15</td>
<td>6</td>
</tr>
<tr>
<td>triangle</td>
<td>5</td>
<td>15</td>
<td>6</td>
</tr>
<tr>
<td>pentagon</td>
<td>5</td>
<td>15</td>
<td>6</td>
</tr>
<tr>
<td>rhombus</td>
<td>5</td>
<td>15</td>
<td>6</td>
</tr>
<tr>
<td>cube</td>
<td>5</td>
<td>15</td>
<td>6</td>
</tr>
<tr>
<td>sphere</td>
<td>5</td>
<td>15</td>
<td>6</td>
</tr>
<tr>
<td>Resource</td>
<td>Resources Permitted</td>
<td>Resources Currently Available</td>
<td>Previous Agent Evaluations</td>
</tr>
<tr>
<td>----------</td>
<td>---------------------</td>
<td>-----------------------------</td>
<td>---------------------------</td>
</tr>
<tr>
<td>circle</td>
<td>5</td>
<td>15</td>
<td>6</td>
</tr>
<tr>
<td>square</td>
<td>5</td>
<td>15</td>
<td>6</td>
</tr>
<tr>
<td>line</td>
<td>0</td>
<td>15</td>
<td>6</td>
</tr>
<tr>
<td>triangle</td>
<td>5</td>
<td>15</td>
<td>6</td>
</tr>
<tr>
<td>pentagon</td>
<td>5</td>
<td>15</td>
<td>6</td>
</tr>
<tr>
<td>rhombus</td>
<td>5</td>
<td>15</td>
<td>6</td>
</tr>
<tr>
<td>cube</td>
<td>5</td>
<td>15</td>
<td>6</td>
</tr>
<tr>
<td>sphere</td>
<td>5</td>
<td>15</td>
<td>6</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Resource</th>
<th>Resources Permitted</th>
<th>Resources Currently Available</th>
<th>Previous Agent Evaluations</th>
</tr>
</thead>
<tbody>
<tr>
<td>circle</td>
<td>5</td>
<td>15</td>
<td>6</td>
</tr>
<tr>
<td>square</td>
<td>5</td>
<td>15</td>
<td>6</td>
</tr>
<tr>
<td>line</td>
<td>5</td>
<td>15</td>
<td>6</td>
</tr>
<tr>
<td>triangle</td>
<td>5</td>
<td>15</td>
<td>6</td>
</tr>
<tr>
<td>pentagon</td>
<td>5</td>
<td>15</td>
<td>6</td>
</tr>
<tr>
<td>rhombus</td>
<td>5</td>
<td>15</td>
<td>6</td>
</tr>
<tr>
<td>cube</td>
<td>5</td>
<td>15</td>
<td>6</td>
</tr>
<tr>
<td>sphere</td>
<td>5</td>
<td>15</td>
<td>6</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Resource</th>
<th>Resources Permitted</th>
<th>Resources Currently Available</th>
<th>Previous Agent Evaluations</th>
</tr>
</thead>
<tbody>
<tr>
<td>circle</td>
<td>5</td>
<td>15</td>
<td>6</td>
</tr>
<tr>
<td>square</td>
<td>5</td>
<td>15</td>
<td>6</td>
</tr>
<tr>
<td>line</td>
<td>5</td>
<td>15</td>
<td>6</td>
</tr>
<tr>
<td>triangle</td>
<td>5</td>
<td>15</td>
<td>6</td>
</tr>
<tr>
<td>pentagon</td>
<td>0</td>
<td>15</td>
<td>6</td>
</tr>
<tr>
<td>rhombus</td>
<td>5</td>
<td>15</td>
<td>6</td>
</tr>
<tr>
<td>cube</td>
<td>5</td>
<td>15</td>
<td>6</td>
</tr>
<tr>
<td>sphere</td>
<td>5</td>
<td>15</td>
<td>6</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Resource</th>
<th>N/W</th>
<th>Min Qty</th>
<th>Rating</th>
</tr>
</thead>
<tbody>
<tr>
<td>circle</td>
<td>N</td>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>square</td>
<td>N</td>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>line</td>
<td>N</td>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>triangle</td>
<td>N</td>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>pentagon</td>
<td>N</td>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>rhombus</td>
<td>N</td>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>cube</td>
<td>N</td>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>sphere</td>
<td>N</td>
<td>3</td>
<td>8</td>
</tr>
</tbody>
</table>
It can be noted that this particular agent required a minimum quantity of three for each of the listed resources, and only Server 4 permitted that quantity of each resource. Using the agent’s requirements, the algorithm correctly selected Server 4.

6.1.2 Test #2: One server has much better resource availability

In this experiment, all of the servers permit and have available a sufficient quantity of each of the resources that the agent needs. In this situation, the algorithm will need to select one server out of a group of servers which can all provide the agent the resources it needs. The major difference among the servers is that Server 5 has fifty of each of the resources currently available, which is much more than any of the other servers. Since all the servers have the same evaluation scores for all the resources, the algorithm is expected to select the server with the greatest quantity of the resources. This decision follows from the fact that the agent is only specifying a minimum quantity of the resources, so there is a possibility that the agent can use more than the minimum.

<table>
<thead>
<tr>
<th>Resource</th>
<th>Server 1</th>
<th>Resources Permitted</th>
<th>Resources Currently Available</th>
<th>Previous Agent Evaluations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Circle</td>
<td>5</td>
<td>10</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>Square</td>
<td>5</td>
<td>10</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>Line</td>
<td>5</td>
<td>10</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>Triangle</td>
<td>5</td>
<td>10</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>Pentagon</td>
<td>5</td>
<td>10</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>rhombus</td>
<td>5</td>
<td>10</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>cube</td>
<td>5</td>
<td>10</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>sphere</td>
<td>5</td>
<td>10</td>
<td>8</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Resource</th>
<th>Server 2</th>
<th>Resources Permitted</th>
<th>Resources Currently Available</th>
<th>Previous Agent Evaluations</th>
</tr>
</thead>
<tbody>
<tr>
<td>circle</td>
<td>5</td>
<td>20</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>square</td>
<td>5</td>
<td>20</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>line</td>
<td>5</td>
<td>20</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>triangle</td>
<td>5</td>
<td>20</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>pentagon</td>
<td>5</td>
<td>20</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>rhombus</td>
<td>5</td>
<td>20</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>cube</td>
<td>5</td>
<td>20</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>sphere</td>
<td>5</td>
<td>20</td>
<td>8</td>
<td></td>
</tr>
</tbody>
</table>
### Server 3

<table>
<thead>
<tr>
<th>Resource</th>
<th>Resources Permitted</th>
<th>Resources Currently Available</th>
<th>Previous Agent Evaluations</th>
</tr>
</thead>
<tbody>
<tr>
<td>circle</td>
<td>5</td>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>square</td>
<td>5</td>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>line</td>
<td>5</td>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>triangle</td>
<td>5</td>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>pentagon</td>
<td>5</td>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>rhombus</td>
<td>5</td>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>cube</td>
<td>5</td>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>sphere</td>
<td>5</td>
<td>3</td>
<td>8</td>
</tr>
</tbody>
</table>

### Server 4

<table>
<thead>
<tr>
<th>Resource</th>
<th>Resources Permitted</th>
<th>Resources Currently Available</th>
<th>Previous Agent Evaluations</th>
</tr>
</thead>
<tbody>
<tr>
<td>circle</td>
<td>5</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>square</td>
<td>5</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>line</td>
<td>5</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>triangle</td>
<td>5</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>pentagon</td>
<td>5</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>rhombus</td>
<td>5</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>cube</td>
<td>5</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>sphere</td>
<td>5</td>
<td>4</td>
<td>8</td>
</tr>
</tbody>
</table>

### Server 5

<table>
<thead>
<tr>
<th>Resource</th>
<th>Resources Permitted</th>
<th>Resources Currently Available</th>
<th>Previous Agent Evaluations</th>
</tr>
</thead>
<tbody>
<tr>
<td>circle</td>
<td>5</td>
<td>50</td>
<td>8</td>
</tr>
<tr>
<td>square</td>
<td>5</td>
<td>50</td>
<td>8</td>
</tr>
<tr>
<td>line</td>
<td>5</td>
<td>50</td>
<td>8</td>
</tr>
<tr>
<td>triangle</td>
<td>5</td>
<td>50</td>
<td>8</td>
</tr>
<tr>
<td>pentagon</td>
<td>5</td>
<td>50</td>
<td>8</td>
</tr>
<tr>
<td>rhombus</td>
<td>5</td>
<td>50</td>
<td>8</td>
</tr>
<tr>
<td>cube</td>
<td>5</td>
<td>50</td>
<td>8</td>
</tr>
<tr>
<td>sphere</td>
<td>5</td>
<td>50</td>
<td>8</td>
</tr>
</tbody>
</table>

### Agent

<table>
<thead>
<tr>
<th>Resource</th>
<th>N/W</th>
<th>Min Qty</th>
<th>Rating</th>
</tr>
</thead>
<tbody>
<tr>
<td>circle</td>
<td>N</td>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>square</td>
<td>N</td>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>line</td>
<td>N</td>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>triangle</td>
<td>N</td>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>pentagon</td>
<td>N</td>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>rhombus</td>
<td>N</td>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>cube</td>
<td>N</td>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>sphere</td>
<td>N</td>
<td>2</td>
<td>7</td>
</tr>
</tbody>
</table>
The agent in this experiment needed the use of resources that were all available on all of the known servers. The algorithm chose Server 5 because it had the greatest available quantity of each of the resources.

6.1.3 Test #3: One server has much better resource evaluations

When an agent specifies a relatively low risk factor, a server with relatively higher evaluations should be selected over any other servers. In this situation, Server 2 has an evaluation value of 9 for all of its resources. The agent is specifying a risk factor of only 10% and the next better server has evaluation values that are all 5, so the algorithm is expected to select Server 2.

<table>
<thead>
<tr>
<th>Resource</th>
<th>Server 1 Resources Permitted</th>
<th>Server 1 Resources Currently Available</th>
<th>Previous Agent Evaluations</th>
</tr>
</thead>
<tbody>
<tr>
<td>circle</td>
<td>4</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>square</td>
<td>4</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>line</td>
<td>4</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>triangle</td>
<td>4</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>pentagon</td>
<td>4</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>rhombus</td>
<td>4</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>cube</td>
<td>4</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>sphere</td>
<td>4</td>
<td>8</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Resource</th>
<th>Server 2 Resources Permitted</th>
<th>Server 2 Resources Currently Available</th>
<th>Previous Agent Evaluations</th>
</tr>
</thead>
<tbody>
<tr>
<td>circle</td>
<td>4</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>square</td>
<td>4</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>line</td>
<td>4</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>triangle</td>
<td>4</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>pentagon</td>
<td>4</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>rhombus</td>
<td>4</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>cube</td>
<td>4</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>sphere</td>
<td>4</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>Resource</td>
<td>Server 3</td>
<td>Resources</td>
<td>Server 4</td>
</tr>
<tr>
<td>----------</td>
<td>----------</td>
<td>-----------</td>
<td>----------</td>
</tr>
<tr>
<td></td>
<td>Resources Permitted</td>
<td>Resources Currently Available</td>
<td>Previous Agent Evaluations</td>
</tr>
<tr>
<td>circle</td>
<td>4</td>
<td>8</td>
<td>3</td>
</tr>
<tr>
<td>square</td>
<td>4</td>
<td>8</td>
<td>3</td>
</tr>
<tr>
<td>line</td>
<td>4</td>
<td>8</td>
<td>3</td>
</tr>
<tr>
<td>triangle</td>
<td>4</td>
<td>8</td>
<td>3</td>
</tr>
<tr>
<td>pentagon</td>
<td>4</td>
<td>8</td>
<td>3</td>
</tr>
<tr>
<td>rhombus</td>
<td>4</td>
<td>8</td>
<td>3</td>
</tr>
<tr>
<td>cube</td>
<td>4</td>
<td>8</td>
<td>3</td>
</tr>
<tr>
<td>sphere</td>
<td>4</td>
<td>8</td>
<td>3</td>
</tr>
</tbody>
</table>

**Agent**

<table>
<thead>
<tr>
<th>Resource</th>
<th>N/W</th>
<th>Min Qty</th>
<th>Rating</th>
</tr>
</thead>
<tbody>
<tr>
<td>circle</td>
<td>N</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>square</td>
<td>N</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>line</td>
<td>N</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>triangle</td>
<td>N</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>pentagon</td>
<td>N</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>rhombus</td>
<td>N</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>cube</td>
<td>N</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>sphere</td>
<td>N</td>
<td>4</td>
<td>8</td>
</tr>
</tbody>
</table>

Risk Factor: 10

Evaluations: 5
In Test #3, Server 2 was chosen due to its outstanding evaluations from previous agents for all of its resources. Since the agent specified a 10% risk, the evaluations were extremely important in the decision. All of the other servers provided the same resources, but they all had much lower evaluations.

6.1.4 Test #4: One server permits much greater quantity of resources

In the following experiment, there are five servers. Server 3 permits a significantly larger quantity of resources than do the other four servers. It is expected that with all the other factors being the same, the algorithm should choose Server 3.

<table>
<thead>
<tr>
<th>Resource</th>
<th>Resources Permitted</th>
<th>Resources Currently Available</th>
<th>Previous Agent Evaluations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Circle</td>
<td>1</td>
<td>20</td>
<td>5</td>
</tr>
<tr>
<td>Square</td>
<td>1</td>
<td>20</td>
<td>5</td>
</tr>
<tr>
<td>Line</td>
<td>1</td>
<td>20</td>
<td>5</td>
</tr>
<tr>
<td>Triangle</td>
<td>1</td>
<td>20</td>
<td>5</td>
</tr>
<tr>
<td>Pentagon</td>
<td>1</td>
<td>20</td>
<td>5</td>
</tr>
<tr>
<td>Rhombus</td>
<td>1</td>
<td>20</td>
<td>5</td>
</tr>
<tr>
<td>Cube</td>
<td>1</td>
<td>20</td>
<td>5</td>
</tr>
<tr>
<td>Sphere</td>
<td>1</td>
<td>20</td>
<td>5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Resource</th>
<th>Resources Permitted</th>
<th>Resources Currently Available</th>
<th>Previous Agent Evaluations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Circle</td>
<td>2</td>
<td>20</td>
<td>5</td>
</tr>
<tr>
<td>Square</td>
<td>2</td>
<td>20</td>
<td>5</td>
</tr>
<tr>
<td>Line</td>
<td>2</td>
<td>20</td>
<td>5</td>
</tr>
<tr>
<td>Triangle</td>
<td>2</td>
<td>20</td>
<td>5</td>
</tr>
<tr>
<td>Pentagon</td>
<td>2</td>
<td>20</td>
<td>5</td>
</tr>
<tr>
<td>Rhombus</td>
<td>2</td>
<td>20</td>
<td>5</td>
</tr>
<tr>
<td>Cube</td>
<td>2</td>
<td>20</td>
<td>5</td>
</tr>
<tr>
<td>Sphere</td>
<td>2</td>
<td>20</td>
<td>5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Resource</th>
<th>Resources Permitted</th>
<th>Resources Currently Available</th>
<th>Previous Agent Evaluations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Circle</td>
<td>10</td>
<td>20</td>
<td>5</td>
</tr>
<tr>
<td>Square</td>
<td>10</td>
<td>20</td>
<td>5</td>
</tr>
<tr>
<td>Line</td>
<td>10</td>
<td>20</td>
<td>5</td>
</tr>
<tr>
<td>Triangle</td>
<td>10</td>
<td>20</td>
<td>5</td>
</tr>
<tr>
<td>Pentagon</td>
<td>10</td>
<td>20</td>
<td>5</td>
</tr>
<tr>
<td>Rhombus</td>
<td>10</td>
<td>20</td>
<td>5</td>
</tr>
<tr>
<td>Cube</td>
<td>10</td>
<td>20</td>
<td>5</td>
</tr>
<tr>
<td>Sphere</td>
<td>10</td>
<td>20</td>
<td>5</td>
</tr>
</tbody>
</table>
The agent needed a minimum quantity of two of each of the listed resources. All the servers had the same evaluation values, and the same resource availabilities. Server 1 only permitted one of each of the resources, so that server should not have been selected. Out of the rest of the servers, Server 3 permitted ten of each resource whereas the rest of the servers permitted five or less of each resource. Server 3 was selected for its greater permission of resource usage.
6.1.5 Test #5: Best resources, but lowest evaluations

This experiment is composed of two separate tests. Using the same server configurations, two different agents use the algorithm to select a server. The first agent is willing to take 0% risk (Agent 0) and the other prefers to take 100% risk (Agent 100). The servers are set up such that Server 1 provides substantially more resources than the rest of the servers, but its evaluation scores from previous agents are all only 1 (extremely low). It is expected that Agent 100 will select Server 1 since it has a much better quantity of resources than the rest of the servers. However, Agent 0 is not expected to select Server 1 because of the low evaluation scores. Agent 0 is expected to select Server 5 since it has the highest evaluation values for all of its resources.

<table>
<thead>
<tr>
<th>Resource</th>
<th>Server 1</th>
<th>Previous Agent Evaluations</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Resources Permitted</td>
<td>Resources Currently Available</td>
</tr>
<tr>
<td>circle</td>
<td>10</td>
<td>50</td>
</tr>
<tr>
<td>square</td>
<td>10</td>
<td>50</td>
</tr>
<tr>
<td>line</td>
<td>10</td>
<td>50</td>
</tr>
<tr>
<td>triangle</td>
<td>10</td>
<td>50</td>
</tr>
<tr>
<td>pentagon</td>
<td>10</td>
<td>50</td>
</tr>
<tr>
<td>rhombus</td>
<td>10</td>
<td>50</td>
</tr>
<tr>
<td>cube</td>
<td>10</td>
<td>50</td>
</tr>
<tr>
<td>sphere</td>
<td>10</td>
<td>50</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Resource</th>
<th>Server 2</th>
<th>Previous Agent Evaluations</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Resources Permitted</td>
<td>Resources Currently Available</td>
</tr>
<tr>
<td>circle</td>
<td>4</td>
<td>10</td>
</tr>
<tr>
<td>square</td>
<td>4</td>
<td>10</td>
</tr>
<tr>
<td>line</td>
<td>4</td>
<td>10</td>
</tr>
<tr>
<td>triangle</td>
<td>4</td>
<td>10</td>
</tr>
<tr>
<td>pentagon</td>
<td>4</td>
<td>10</td>
</tr>
<tr>
<td>rhombus</td>
<td>4</td>
<td>10</td>
</tr>
<tr>
<td>cube</td>
<td>4</td>
<td>10</td>
</tr>
<tr>
<td>sphere</td>
<td>4</td>
<td>10</td>
</tr>
</tbody>
</table>
### Server 3

<table>
<thead>
<tr>
<th>Resource</th>
<th>Resources Permitted</th>
<th>Resources Currently Available</th>
<th>Previous Agent Evaluations</th>
</tr>
</thead>
<tbody>
<tr>
<td>circle</td>
<td>4</td>
<td>10</td>
<td>3</td>
</tr>
<tr>
<td>square</td>
<td>4</td>
<td>10</td>
<td>3</td>
</tr>
<tr>
<td>line</td>
<td>4</td>
<td>10</td>
<td>3</td>
</tr>
<tr>
<td>triangle</td>
<td>4</td>
<td>10</td>
<td>3</td>
</tr>
<tr>
<td>pentagon</td>
<td>4</td>
<td>10</td>
<td>3</td>
</tr>
<tr>
<td>rhombus</td>
<td>4</td>
<td>10</td>
<td>3</td>
</tr>
<tr>
<td>cube</td>
<td>4</td>
<td>10</td>
<td>3</td>
</tr>
<tr>
<td>sphere</td>
<td>4</td>
<td>10</td>
<td>3</td>
</tr>
</tbody>
</table>

### Server 4

<table>
<thead>
<tr>
<th>Resource</th>
<th>Resources Permitted</th>
<th>Resources Currently Available</th>
<th>Previous Agent Evaluations</th>
</tr>
</thead>
<tbody>
<tr>
<td>circle</td>
<td>4</td>
<td>10</td>
<td>4</td>
</tr>
<tr>
<td>square</td>
<td>4</td>
<td>10</td>
<td>4</td>
</tr>
<tr>
<td>line</td>
<td>4</td>
<td>10</td>
<td>4</td>
</tr>
<tr>
<td>triangle</td>
<td>4</td>
<td>10</td>
<td>4</td>
</tr>
<tr>
<td>pentagon</td>
<td>4</td>
<td>10</td>
<td>4</td>
</tr>
<tr>
<td>rhombus</td>
<td>4</td>
<td>10</td>
<td>4</td>
</tr>
<tr>
<td>cube</td>
<td>4</td>
<td>10</td>
<td>4</td>
</tr>
<tr>
<td>sphere</td>
<td>4</td>
<td>10</td>
<td>4</td>
</tr>
</tbody>
</table>

### Server 5

<table>
<thead>
<tr>
<th>Resource</th>
<th>Resources Permitted</th>
<th>Resources Currently Available</th>
<th>Previous Agent Evaluations</th>
</tr>
</thead>
<tbody>
<tr>
<td>circle</td>
<td>4</td>
<td>10</td>
<td>8</td>
</tr>
<tr>
<td>square</td>
<td>4</td>
<td>10</td>
<td>8</td>
</tr>
<tr>
<td>line</td>
<td>4</td>
<td>10</td>
<td>8</td>
</tr>
<tr>
<td>triangle</td>
<td>4</td>
<td>10</td>
<td>8</td>
</tr>
<tr>
<td>pentagon</td>
<td>4</td>
<td>10</td>
<td>8</td>
</tr>
<tr>
<td>rhombus</td>
<td>4</td>
<td>10</td>
<td>8</td>
</tr>
<tr>
<td>cube</td>
<td>4</td>
<td>10</td>
<td>8</td>
</tr>
<tr>
<td>sphere</td>
<td>4</td>
<td>10</td>
<td>8</td>
</tr>
</tbody>
</table>

### Agent 0

<table>
<thead>
<tr>
<th>Resource</th>
<th>N/W</th>
<th>Min Qty</th>
<th>Rating</th>
</tr>
</thead>
<tbody>
<tr>
<td>circle</td>
<td>N</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>square</td>
<td>N</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>line</td>
<td>N</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>triangle</td>
<td>N</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>pentagon</td>
<td>N</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>rhombus</td>
<td>N</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>cube</td>
<td>N</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>sphere</td>
<td>N</td>
<td>3</td>
<td>6</td>
</tr>
</tbody>
</table>
Again, the algorithm worked just as expected. Agent 0 selected Server 5 due to its higher evaluation scores from previous agents and the agent’s risk of 0%. Agent 100 selected Server 1 due to its substantially greater quantities of resources (even though it had very poor evaluations) and the agent’s willingness to accept 100% risk.

6.1.6 Test #6: Perfect evaluations, much fewer resources

Similar to Test #5, two agents (Agent 0 and Agent 100) are used in this experiment as well. The servers in this experiment are Server 3, which has perfect evaluations for all of its resources, and four other servers which have evaluation scores of 6 for all of their resources. The other major factor is that Server 3 has the lowest quantities of its resources, and Server 4 has the greatest quantities of its resources. Agent 0 is expected to select Server 3 and Agent 100 is expected to select Server 4 due to its high resource availability.
<table>
<thead>
<tr>
<th>Resource</th>
<th>Resources Permitted</th>
<th>Resources Currently Available</th>
<th>Previous Agent Evaluations</th>
</tr>
</thead>
<tbody>
<tr>
<td>circle</td>
<td>4</td>
<td>20</td>
<td>6</td>
</tr>
<tr>
<td>square</td>
<td>4</td>
<td>20</td>
<td>6</td>
</tr>
<tr>
<td>line</td>
<td>4</td>
<td>20</td>
<td>6</td>
</tr>
<tr>
<td>triangle</td>
<td>4</td>
<td>20</td>
<td>6</td>
</tr>
<tr>
<td>pentagon</td>
<td>4</td>
<td>20</td>
<td>6</td>
</tr>
<tr>
<td>rhombus</td>
<td>4</td>
<td>20</td>
<td>6</td>
</tr>
<tr>
<td>cube</td>
<td>4</td>
<td>20</td>
<td>6</td>
</tr>
<tr>
<td>sphere</td>
<td>4</td>
<td>20</td>
<td>6</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Resource</th>
<th>Resources Permitted</th>
<th>Resources Currently Available</th>
<th>Previous Agent Evaluations</th>
</tr>
</thead>
<tbody>
<tr>
<td>circle</td>
<td>4</td>
<td>5</td>
<td>10</td>
</tr>
<tr>
<td>square</td>
<td>4</td>
<td>5</td>
<td>10</td>
</tr>
<tr>
<td>line</td>
<td>4</td>
<td>5</td>
<td>10</td>
</tr>
<tr>
<td>triangle</td>
<td>4</td>
<td>5</td>
<td>10</td>
</tr>
<tr>
<td>pentagon</td>
<td>4</td>
<td>5</td>
<td>10</td>
</tr>
<tr>
<td>rhombus</td>
<td>4</td>
<td>5</td>
<td>10</td>
</tr>
<tr>
<td>cube</td>
<td>4</td>
<td>5</td>
<td>10</td>
</tr>
<tr>
<td>sphere</td>
<td>4</td>
<td>5</td>
<td>10</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Resource</th>
<th>Resources Permitted</th>
<th>Resources Currently Available</th>
<th>Previous Agent Evaluations</th>
</tr>
</thead>
<tbody>
<tr>
<td>circle</td>
<td>4</td>
<td>80</td>
<td>6</td>
</tr>
<tr>
<td>square</td>
<td>4</td>
<td>80</td>
<td>6</td>
</tr>
<tr>
<td>line</td>
<td>4</td>
<td>80</td>
<td>6</td>
</tr>
<tr>
<td>triangle</td>
<td>4</td>
<td>80</td>
<td>6</td>
</tr>
<tr>
<td>pentagon</td>
<td>4</td>
<td>80</td>
<td>6</td>
</tr>
<tr>
<td>rhombus</td>
<td>4</td>
<td>80</td>
<td>6</td>
</tr>
<tr>
<td>cube</td>
<td>4</td>
<td>80</td>
<td>6</td>
</tr>
<tr>
<td>sphere</td>
<td>4</td>
<td>80</td>
<td>6</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Resource</th>
<th>Resources Permitted</th>
<th>Resources Currently Available</th>
<th>Previous Agent Evaluations</th>
</tr>
</thead>
<tbody>
<tr>
<td>circle</td>
<td>4</td>
<td>50</td>
<td>6</td>
</tr>
<tr>
<td>square</td>
<td>4</td>
<td>50</td>
<td>6</td>
</tr>
<tr>
<td>line</td>
<td>4</td>
<td>50</td>
<td>6</td>
</tr>
<tr>
<td>triangle</td>
<td>4</td>
<td>50</td>
<td>6</td>
</tr>
<tr>
<td>pentagon</td>
<td>4</td>
<td>50</td>
<td>6</td>
</tr>
<tr>
<td>rhombus</td>
<td>4</td>
<td>50</td>
<td>6</td>
</tr>
<tr>
<td>cube</td>
<td>4</td>
<td>50</td>
<td>6</td>
</tr>
<tr>
<td>sphere</td>
<td>4</td>
<td>50</td>
<td>6</td>
</tr>
</tbody>
</table>
Agent 0 specified a 0% risk when selecting a server, so the algorithm selected Server 3 with perfect evaluation scores (even though it had extremely low resource quantities). Agent 100 selected Server 4 since it had the greatest quantity of available resources.
### 6.1.7 Review of results

Below is a table that reviews the results from all the boundary case experiments.

<table>
<thead>
<tr>
<th><strong>Experiment</strong></th>
<th><strong>Agent Name</strong></th>
<th><strong>Server Chosen</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Test #1: Only Server 4 had all the necessary resources. Each of the other servers doesn't permit access to one of the resources.</td>
<td>Agent: Needed 3 of each resource with 10% risk.</td>
<td>Server 4: The only server that permitted at least 3 of each resource.</td>
</tr>
<tr>
<td>Test #2: Server 5 had much better resource availability than the other servers did. All servers had the same evaluation scores.</td>
<td>Agent: Needed 2 of each resource with 10% risk.</td>
<td>Server 5: Had 50 of each resource currently available.</td>
</tr>
<tr>
<td>Test #3: Server 2 had much better resource evaluations (9) than the rest (&lt; 6). Resource availabilities and permissions were the same.</td>
<td>Agent: Needed 4 of each resource with 10% risk.</td>
<td>Server 2: Had evaluation scores of all 9's.</td>
</tr>
<tr>
<td>Test #4: Server 3 permitted a greater quantity of resources (10 each) than the rest (&lt; 6 each). The permissions and evaluations were the same.</td>
<td>Agent: Needed 2 of each resource with 10% risk.</td>
<td>Server 3: Permitted access to 10 of each of the resources.</td>
</tr>
<tr>
<td>Test #5: Best resources, but lowest evaluations. Server 1 had the best availability and permitted quantities of resources, but Server 5 had the best evaluation scores (8 as opposed to &lt; 5).</td>
<td>Agent 0: Needed 3 of each resource, with a risk of 0%.</td>
<td>Server 5: Permitted and had available the same quantities of resources as all the servers, except for Server 1, but Server 5 had the best evaluations of all the servers.</td>
</tr>
<tr>
<td>Test #5: Best resources, but lowest evaluations. Server 1 had the best availability and permitted quantities of resources, but Server 5 had the best evaluation scores (8 as opposed to &lt; 5).</td>
<td>Agent 100: Needed 3 of each resource, with a risk of 100%.</td>
<td>Server 1: Permitted and had available the largest quantity of resources, but had the lowest evaluation scores of all the servers (all 1's).</td>
</tr>
<tr>
<td>Test #6: Perfect evaluations, much fewer resources. Server 3 was the only server to have perfect evaluations for all of its resources (all 10's). All servers permitted the same amount of resources. Server 4 had substantially more resources available than any other server did.</td>
<td>Agent 0: Needed 4 of each resource, with a risk of 0%</td>
<td>Server 3: The only server with perfect evaluations for all of its resources, but it had the lowest availability of resources.</td>
</tr>
<tr>
<td>Test #6: Perfect evaluations, much fewer resources. Server 3 was the only server to have perfect evaluations for all of its resources (all 10's). All servers permitted the same amount of resources. Server 4 had substantially more resources available than any other server did.</td>
<td>Agent 100: Needed 4 of each resource, with a risk of 100%</td>
<td>Server 4: Had the most available resources of all the servers, and had the same evaluations as all the servers, except for Server 3.</td>
</tr>
</tbody>
</table>

**Table #1: Review of all test results**
6.2 Print Job Scheduling Simulations

Before the algorithm could be used in a print job scheduling environment, the agents using the algorithm needed to know a little bit about printing. It is assumed that a server exists (the information server) that would provide current information about the features and states of all the printers in the cluster. The agents would need to take the information pertinent to printing and fit it into the inputs to the algorithm.

The first inputs to the algorithm are the importance values for each of the resources in which the agent is interested. So, what information regarding printing should be considered a resource in this algorithm? The easy answers to this question are the features of common printers such as duplex capability, stapling capability, containing colored paper, etc. DPI was chosen to be a printing feature that must be specified for the algorithm to work. A minimum DPI must be specified and it is regarded as a necessary feature.

The algorithm requires that the printers permit some features to be used by certain users and that printers declare a particular quantity of each feature that is currently available. The assumption was made that if a printer has a particular feature, then it is currently available and that all users are allowed to use it. Following these assumptions, the algorithm was simplified such that the availability and permission matrices are replaced by a single printer feature matrix. If a printer has a feature, then a one is placed in the appropriate position in the matrix for that printer and that feature.

Another important factor in printing is how soon a document can be printed. The user is allowed to specify a minimum time before which the user needs or desires the document to be printed. If the user specifies that the document must be printed within five minutes, but none of the printers are capable of providing that service, the algorithm will respond by saying that the job could not be scheduled. An assumption was made that jobs cannot be divided among several printers in the cluster. Whether the user specifies that the minimum time is a need or that it's just wanted, the user is still required to provide an importance factor for the time. The only way to decide if a job can be printed within the necessary amount of time is to know how many pages are in the document and the speed of the printers. It is assumed that all documents are being
printed on letter size paper and that all documents can be printed at the full speed of the printers.

Once a printer is decided to be fast enough to print the user’s document, a value must be placed in the printer features matrix to represent the printer’s speed. Just placing the speed of the printer in the matrix is not sufficient. It is necessary is to use a value that represents how soon the printer would be able to print the new job (assuming no jams). The following equation was used to come up with a value for the matrix:

\[ v = \frac{x \cdot s}{x + x_0} \]

The \( v \) represents the score for the printer that will be placed in the matrix. \( s \) represents the speed of the printer in pages per minute. \( x \) represents the size of the new job (in pages) to be scheduled, and \( x_0 \) represents the number of pages currently queued on the particular printer. Using this equation provides a score that is inversely proportional the amount of time the printer should take to finish its current work and to print the new job.

If jobs were allowed to be divided among several printers, clusters of printers would receive their own positions in each of the matrices. A printer cluster would have the intersection of the features of each of the member printers. Column vectors would be formed for each of the printers (using only the features in the intersection) for each of the matrices of information about a printer (features, and previous agent evaluations). The sum of the column vectors for the features information could be placed in the appropriate matrix. However, the evaluations should not be merely added. Instead, the evaluation scores from each printer should be averaged in order to form one value for each feature. This solution should work except for a little extra modification to allow for the uneven division of the work among the printers. This modification would probably take into account the score from the equation above in order to schedule more work on printers with higher scores.

The last major piece of information is the evaluations. It is assume that after a job has been completely printed, the agent responsible for that print job leaves evaluations with the information server. If the job prints on time, the printer would receive an average evaluation. If a printer completes a job much faster than was needed, the printer would probably receive a very good evaluation, and if a printer is very late, the printer
would receive a poor evaluation. But this information only involves the speed of the printer, so there was an extra evaluations category added to the algorithm. The "OnTime" category is associated with the feature score that is calculated from the equation above. The agent would evaluate each of the other resources used as well. Assuming an advanced printing system, the agent would be able to find out if the printer had run out of red paper during its job, or if the stapler jammed at some time. This type of information would allow the agent to evaluate each of the other resources that it used.

Then, the agent would require an algorithm to look at previous evaluations of printers in order to come up with a value for each feature for the evaluations matrix. The evaluations were assumed available and that some other server was assumed to be providing the single value representing the evaluations.

Since evaluations are available, the users submitting jobs to the cluster can specify a risk factor for each job. When a user submits a job with a low risk factor, the algorithm will only consider using printers with relatively good evaluations.

6.2.1 Simulation #1: Only one printer has all the necessary features

This experiment involves a two-printer environment in which a user submits fifteen sequential jobs of the same document and the same settings. Only Printer 1 has all of the features that the jobs require. Even though Printer 2 is substantially faster than Printer 1, all of the following jobs should be scheduled on Printer 1. Both printers begin with some pages in their queues. All of the new jobs should be scheduled to print on Printer 1 because of the job settings. The specifications for the two printers are:

<table>
<thead>
<tr>
<th>Feature</th>
<th>Printer 1</th>
<th>Printer 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Speed (pages/min)</td>
<td>40</td>
<td>1000</td>
</tr>
<tr>
<td>DPI</td>
<td>600</td>
<td>1200</td>
</tr>
<tr>
<td>Currently Queued Pages</td>
<td>50</td>
<td>1500</td>
</tr>
</tbody>
</table>
The current evaluations for each of the features of the two printers are:

<table>
<thead>
<tr>
<th>Feature</th>
<th>Printer 1</th>
<th>Printer 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Duplex</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>Binder</td>
<td></td>
<td>8</td>
</tr>
<tr>
<td>Staple</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>Blue Paper</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>White Paper</td>
<td></td>
<td>8</td>
</tr>
<tr>
<td>Red Paper</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>OnTime</td>
<td>8</td>
<td>8</td>
</tr>
</tbody>
</table>

The features of the job that is repeatedly submitted are:

<table>
<thead>
<tr>
<th>Risk</th>
<th>DPI</th>
<th>Pages</th>
<th>Need/Want</th>
<th>Duplex</th>
<th>Staple</th>
<th>Min Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>10%</td>
<td>600</td>
<td>100</td>
<td>Needed</td>
<td>8</td>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Needed</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Rating:</td>
<td>8</td>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Minimum:</td>
<td>N/A</td>
<td>N/A</td>
<td>10 min</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

One Usable Printer

Figure #1: Simulation of one usable printer
It should be noted that on this graph Printer 2 only finishes printing the pages that were in its queue. All of the new jobs are scheduled on Printer 1 (causing its queue to slowly become more full) because only Printer 1 has the necessary features for the new jobs. This is exactly what was expected to happen in the simulation.

6.2.2 Simulation #2: Same job to similar printers

In this experiment, both Printer 1 and Printer 2 have the same features and the same evaluations for each of those features. Both printers have the same specifications and both start out with empty queues. The user submits thirty-five jobs of the same document with the same settings to the printer cluster. Between each submission there is a small pause to let the printers output a few pages. In this situation, the jobs should be scheduled on alternating printers since one printer will always have fewer pages currently queued than the other does (except at the beginning when both printers are idle). The specifications for the two printers are:

<table>
<thead>
<tr>
<th></th>
<th>Printer 1</th>
<th>Printer 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Speed (pages/min)</td>
<td>40</td>
<td>40</td>
</tr>
<tr>
<td>DPI</td>
<td>600</td>
<td>600</td>
</tr>
<tr>
<td>Currently Queued Pages</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

The current evaluations for each of the features of the two printers are:

<table>
<thead>
<tr>
<th></th>
<th>Printer 1</th>
<th>Printer 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Duplex</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>Staple</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>OnTime</td>
<td>8</td>
<td>8</td>
</tr>
</tbody>
</table>

The features of the job that is repeatedly submitted are:

<table>
<thead>
<tr>
<th>Risk</th>
<th>DPI</th>
<th>Pages</th>
<th>Duplex</th>
<th>Staple</th>
<th>Min Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>10%</td>
<td>600</td>
<td>100</td>
<td>Need</td>
<td>Need</td>
<td>Want</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Rating</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>8</td>
<td>2</td>
<td></td>
<td>10 min</td>
</tr>
<tr>
<td></td>
<td>N/A</td>
<td>N/A</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Same Job to Similar Printers

Figure #2: Simulation of submitting the same job to similar printers

This graph demonstrates that with both queues initially empty, the algorithm simply selected the first printer, Printer 1. All the following jobs alternated between the two printers. Since the printers were given a small amount of time to print between each new submission, alternating printers had fewer pages currently queued at the time of each successive job submission. This simulation provided exactly the results that were expected.

6.2.3 Simulation #3: Same job to printers with changing evaluations

In this experiment, both Printer 1 and Printer 2 have the same features and the same evaluations for each of those features. Both printers have the same specifications and both start out with empty queues. The same job is repeatedly submitted to the printer cluster. After each submission, the printers are given a small amount of time to print a few pages before the next job submission. This experiment differs from the previous in that during this experiment, the evaluation scores for the printers will be artificially modified throughout the experiment.
This experiment consists of two separate tests. One test submits jobs with a risk factor of 30% and the other test uses 80% risk. Each test will involve one thousand jobs being submitted to the scheduler, allowing a small amount of printing to take place between submissions. For each test, after the first thirty print jobs are submitted, the evaluation scores for Printer 1 will artificially be forced to decrease until the total evaluation score is half of the total score of Printer 2. After both printers remain in that state for a little while, the evaluation scores for Printer 2 will slowly be artificially forced to decrease until they reach zero.

The jobs should tend to avoid the printer with a significantly lower evaluation rating (being “significantly lower” will depend upon the risk factor specified for that set of jobs).

The specifications for the two printers are:

<table>
<thead>
<tr>
<th></th>
<th>Printer 1</th>
<th>Printer 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Speed (pages / min)</td>
<td>40</td>
<td>40</td>
</tr>
<tr>
<td>DPI</td>
<td>600</td>
<td>600</td>
</tr>
<tr>
<td>Currently Queued Pages</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

The current evaluations for each of the features of the two printers are:

<table>
<thead>
<tr>
<th></th>
<th>Printer 1</th>
<th>Printer 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Duplex</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>Staple</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>OnTime</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

The features of the job that is repeatedly submitted are:

<table>
<thead>
<tr>
<th>DPI</th>
<th>Pages</th>
<th>Need/Want:</th>
<th>DUPLEX</th>
<th>STAPLE</th>
<th>MIN Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>600</td>
<td>100</td>
<td>Need/Need:</td>
<td>8</td>
<td>8</td>
<td>Want 10 min</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Rating:</th>
<th>Minimum:</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>N/A</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Minimum:</th>
</tr>
</thead>
<tbody>
<tr>
<td>10 min</td>
</tr>
</tbody>
</table>
Changing Printer Evaluations with 30\% Risk

Figure #3: Simulation of changing printer evaluations with 30\% risk

Changing Printer Evaluations with 80\% Risk

Figure #4: Simulation of changing printer evaluations with 80\% risk
In the first graph, Printer 2 had some time during which jobs were scheduled only on it, but then around time=3000s the jobs began favoring Printer 1. On the second graph, with an 80% risk, both printers were equally favored for a much greater amount of time. It wasn’t until around time=4250s that Printer 1 was favored over Printer 2 and by that time the evaluations for Printer 2 were very close to 0. These are the results that were expected from the simulation.

6.2.4 Simulation #4: Different printers with changing job specifications

In this experiment, Printer 1 and Printer 2 have different features. However, both printers have the same specifications and both start out with empty queues. Jobs will be repeatedly submitted to the printer cluster. Throughout the experiment, the job settings will remain the same, but the importance of those settings, to the agent, will vary. After each submission, the printers are given enough time to print all the pages in their queues prior to the next job submission.

This experiment is composed of three separate tests where each uses a different risk factor for all the submitted jobs. The risk factors for each of the tests are 10%, 20%, and 90% respectively.

Each of the three tests will include one thousand print jobs. After the first thirty print jobs, the desire rating for the 3-holed paper will begin to slowly decrease from an initial value of 10 until it reaches zero. Then, the rating for 3-holed paper will be reset to 5, and the rating for the red paper will be reset to 10. For the next set of jobs, the rating for the red paper will slowly decrease to zero where it will remain for the rest of the jobs.

The specifications for the two printers are:

<table>
<thead>
<tr>
<th></th>
<th>Printer 1</th>
<th>Printer 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Speed (pages / min)</td>
<td>1000</td>
<td>1000</td>
</tr>
<tr>
<td>DPI</td>
<td>600</td>
<td>600</td>
</tr>
<tr>
<td>Currently Queued Pages</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
The current evaluations for each of the features of the two printers are:

<table>
<thead>
<tr>
<th>Feature</th>
<th>Printer 1</th>
<th>Printer 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Red Paper</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>3-holed Paper</td>
<td></td>
<td>10</td>
</tr>
<tr>
<td>Staple</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>OnTime</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

The initial features of the job that is repeatedly submitted are:

<table>
<thead>
<tr>
<th>DPI</th>
<th>Pages</th>
<th>3-Holed</th>
<th>Red</th>
<th>Staple</th>
<th>Min Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>600</td>
<td>10</td>
<td>Need/Want:</td>
<td>Want</td>
<td>Want</td>
<td>Rating: 10</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Minimum:</td>
<td>N/A</td>
<td>N/A</td>
<td>Minimum: N/A</td>
</tr>
</tbody>
</table>

**Changing Jobs at 10% Risk**

Figure #5: Simulation of changing job feature importance factors at 10% risk
Changing Jobs at 20% Risk

Figure #6: Simulation of changing job feature importance factors at 20% risk

Changing Jobs at 90% Risk

Figure #7: Simulation of changing job feature importance factors at 20% risk
The most important data series to focus on is the one for Printer 1 in all three graphs. It should be noted that the line for Printer 1 at the value of 10 gets longer as the risk factor gets larger. The importance of this experiment is that it demonstrates the effect of varying importance factors for different features.

Remember that Printer 1 has red paper (no 3-holed) with an evaluation rating of 7, and Printer 2 has 3-holed paper (no red) with an evaluation rating of 10. Analyzing the 10% risk graph first, it is noticed that a much greater drop in importance of 3-holed paper was needed for the algorithm to stop selecting Printer 2. Since the red paper has a lower evaluation score, the algorithm required only a small decrease in importance of red paper in order to stop selecting Printer 1. The same trend can be seen in the 20% risk graph, but the difference in importance factors necessary to cause the change of selection is greatly decreased (from about 7 at 10% to about 2 for 20%). At 90% risk, Printer 1 is always selected because the algorithm will essentially choose the printer with the best score for the features. Since both printers have the one necessary feature, and each has one of the desired features, they both end up with the same feature score. Therefore, the algorithm just selects the first printer in the list (Printer 1).

The results of this simulation were essentially what was expected. The only surprise was the drastic effect that the risk factor had on the selection. The large difference between the 10% and 20% graphs was a small surprise, but looking at it now, it makes sense due to the very small number of features that were being used. If more features had been used in the simulation, the effect of the red paper and the 3-holed paper would have been diluted.

6.2.5 Review of results

Each of the simulations produced results that were expected. The only surprise was in Simulation #4 in which the importance factors were varied for two job features. The risk factor was expected to have a strong effect on the selection, but the amount of effect that the risk factor had was surprising.
7 Future Work

7.1 Specifying a Server Preference

One enhancement of the algorithm that is much needed is the ability for an agent to specify server preferences. Assuming the algorithm could be used in one of two ways, the algorithm can execute on a server or as a function in an agent. If the algorithm is run on a centralized server, then there need to be inputs to the algorithm in order to specify preferred servers and ratings of those servers. If the algorithm is being run by the agent, then the agent can just modify the last few steps of the algorithm to weight certain servers more than others.

As agents become more intelligent and it becomes less of a concern as to how much baggage an agent carries around, the idea of an agent maintaining a list of preferred servers becomes much more reasonable. The ability to specify preferences in servers is a very important feature that should be added to the algorithm.

A difficulty that would need to be resolved is what an agent should do if a user specifies a particular preference, but the agent’s experience conflicts with the user’s request. An agent’s responsibility is to perform tasks for a user. When it comes time to use the user’s recommendation, it might not be possible to contact the user to ask for further guidance. What should the agent do? The agent is expected to perform the task. Maybe a user interface to the agent provides the options to make these specifications. This needs to be resolved.

7.2 Different Weights for Different Evaluations

There is an inherent problem with the algorithm if it is to be used outside of a very close-knit local network. If it is possible to consider servers that have been evaluated by completely unknown agents, then there is a big question as to how much those other agents should be trusted. It would make sense that some agents wouldn’t think too highly of evaluations from other particular agents. It is the same way that humans act; some people trust certain people more than they trust other people. Independent of trust, some people have similar interests as other people. If an agent
needs to print something, the agent might trust the evaluations of one group of agents, but if the agent needs access to a database, it might prefer evaluations from a different group of agents.

Giving agents the option to specify that some agents are trusted more than others is a very important feature that would be needed in a large network. Perhaps agents can be grouped in different ways. Each agent should be able to define the boundaries for different groups and there could even be globally recognized groups that agents could specify.

An agent should be able to specify varying degrees of usefulness of different agents or groups of agents for different resources. Similar to the importance factors that an agent must specify for each resource, an agent could also list specific agents or agent groups for each resource. For each agent or group, the agent could optionally specify the value of any information provided by the other agent(s). Perhaps a range of [0..10] would be acceptable. There is a problem if an agent is specifically listed as well as a group to which it belongs, or if multiple groups with an intersection other than null are listed. These problems would need to be resolved. If an agent is going to carry lists of other agents' identities, when does that amount of baggage become too cumbersome to carry?

7.3 Different Types of Evaluations

Just as the previous topic had been mentioned in Chapter 5, so had this one. The idea behind this future enhancement is that a single value for an evaluation of a resource provides minimal information to future agents. Perhaps this would be sufficient in a local network situation, but it is assumed that as the use of this algorithm is scaled to larger networks, the information is insufficient.

A strength of the algorithm is that there is not a defined set of rules that agents must follow when determining an evaluation value. This happens to be a weakness of the algorithm as well. As the number of different agents increases, it will become more necessary to provide more specific information as to why a particular server received each score.
The goal of this extended research would involve the discovery of a good set of factors that an agent can evaluate about each resource. The factors would need to be specific enough to give other agents some idea about what the value is in regards to, but the factors should also be general enough to provide the agents with enough flexibility and individuality. This is a tough question that should be answered.

7.4 Evaluation Algorithms

Related to the previous two sections is the problem of creating different types of agents that evaluated servers differently. Just as different people evaluate things differently from one another, agents should do the same. Different types of algorithms need to be developed and tested in different environments. There will probably end up being different evaluation algorithms that are best suited for particular types of resources.

The first problem that needs to be resolved is what are some different algorithms that can be used. Secondly, the algorithms need to be analyzed to find out their strengths and weaknesses. Lastly, an analysis needs to be performed to find out how an agent can decide which algorithms to bring with it on particular trips. The last item involves the fact that agents can’t bring all of their tools with them everywhere they go. Sometimes an agent will have to use an algorithm that is not the best-suited one for a particular resource, but the agent couldn’t bring the preferred algorithm. Before the agent migrates, it would need to decide on the best set of algorithms to bring with it for all the resources that it plans on evaluating. The previous statement implies that one option for an agent is to not evaluate resources for which it doesn’t have the optimum evaluation algorithm.

There is a lot of research that can be done in this area.

7.5 Handling Dynamic Resource Availability

The availability of various resources on a particular server can be quite dynamic. The evaluations are a tool to help agents decide whether or not it is good to migrate to a particular server. By placing a time stamp on evaluations, future agents can try to guess as to when good times are to go to particular servers.

Choosing an agent server can be very similar to choosing stocks in which to invest. There are often trends to be discovered in the usage of various resources on particular servers. If there is sufficient information available to agents, they can try to
find the trends. It is impossible to foretell the future, but if server usage follows patterns then an agent will have a better chance at choosing a good server.

It will be important to develop various algorithms for discovering resource usage patterns on servers. Along with this investigation, there will be questions that need to be answered. How much information will an agent need from a server to make an intelligent decision? Will migrating agents perform the pattern discovery themselves or will there be dedicated servers/agents that will provide pattern information to agents considering migration? What types of evaluations data will be required to perform a good analysis?

When performing the analysis, it will be important to look at long term and short term trends. The trends will involve analysis as to whether more resources are becoming available or if they are declining, and it is important to know at what rate they are changing. Then the agent needs to recognize whether it will be using the resources for a long time or a very short period. This could be a difficult decision by itself.

7.6 Non-Linear Resource Ranking

This topic was mentioned in Chapter 5, but deserves further comment in this chapter. Not all resources are linear in importance as the available quantity increases.

A simple example is RAM verses disk space. If an algorithm requires a lot of temporary work space, then usually disk space will be used. However, in order to perform the algorithm at all a minimum amount of RAM is necessary. In this scenario, the agent would probably specify that the minimum RAM is necessary, but wouldn’t give it too high of an importance factor. The agent probably really would like to specify that if a server has another specified amount of RAM available, then the RAM would become more important to the agent. Having a certain amount of RAM available would allow the agent to use the RAM to dramatically improve the speed of the algorithm. Perhaps, if another level of RAM is available, the importance of the RAM would become even more important.

Another similar situation would involve a scenario such that an agent needs two of three specified resources. An agent might want to specify that the combination of the first and third resources would be most desired, and the combination of the second and third resources would be least desired. The combination of the first and second resources
would have an importance somewhere between the other two combinations. This example could be scaled up to many different resources where some combinations are valid and others aren’t, and the ones that are valid might have different importance values.

There is a need to be able to specify the importance of resources in a much more flexible and powerful manner. The current algorithm design only allows for simple specifications of importance of individual resources. There would need to be research done on the topics of different means of specifying resource importance in a more powerful manner, and methods of incorporating the information into the selection algorithm.

7.7 Universal Naming Scheme

In order for this algorithm to be widely accepted and usable there is a need to specify rules for naming resources. There are many different types of resources and there are often many different names for the same type of resource. There are also many resources that can be classified in different hierarchical groupings. In order for an agent to locate resources on various servers, there needs to be an accepted naming scheme for all resources.

An agent might be searching for a server with a license for “SuperCad”, but all the servers refer to the software as “Super-Cad”. This could cause a big problem for the agent since there are probably licenses available for the software, but it will never find them because of a naming inconsistency. This problem could be solved if all servers and agents followed the same naming scheme for resources.

Another agent might be searching for a server with an accessible modem attached to it. A problem here is that a server might refer to the modem by the make and model of the actual device. Another server might refer to the modem by its capable protocol versions. Yet another server might just refer to the modem as a “modem”. The agent would be able to find each of the servers if there was a naming scheme that provided all of the information that the servers provided individually. Having all of the previously mentioned information would allow an agent to look for a device by referring to it with a generic name, or by the device’s specific make and model, or by its characteristics.
A similar problem is the hierarchical grouping of resources. The group of “Output Devices” could be divided into both visual and audio output devices. Then each of those groups could be further subdivided until there is a full set of information describing every device. This would allow agents to specify that they are looking for a video display and the agent should be able to find CRT displays, LCD displays, overhead projectors, etc.

If many different agents are going to be able to interact with many different types of servers, there needs to be a common language among them.

7.8 Resource List from Task Description

A difficult task for an agent is to be able to understand the true desires of the human using the agent. Humans typically communicate using languages that require experience, intuition, reasoning, and assumptions in order for someone to understand the language. There are many people who would prefer to communicate with computers using their own native human language. There has been much research conducted in the area of natural language comprehension by computers, but with the advent of agents there is a renewed interest in having computers capable of understanding natural languages.

It would be very useful if a user could write a description (in a natural language) of a set of tasks for an agent, and then expect the agent to perform the tasks. Often humans take for granted the fact that other humans understand that certain pre-conditions must be met before certain tasks can be accomplished, so this is usually taken for granted when giving instructions to other people. It is very difficult for a computer to realize pre-conditions for tasks. Another associated difficulty is the comprehension of the set of necessary computer resources that tasks imply. An example of this would be a person instructing an agent to deliver a message to another person. The agent would need to realize that this instruction implies locating a means of communication between the local computer and the device with which the other person is going to receive the message. This ability will be very important in future uses of agents.
7.9 Deciding on Risk Factor and Resource Importance

Deciding on risk factors is not an easy problem. It might be easy for humans sometimes to just always specify a particular risk factor, but it would probably be more useful if an agent could modify the risk factor depending on the particular task. Another closely related difficulty is the determination of resource importance for particular tasks. It is desired to be able to just give an agent a list of tasks that need to be done and maybe a specification of time frames and importance of each task.

An agent should be able to receive a list of tasks and be able to accomplish them with very little human intervention. Having an overall time frame for the set of tasks would sometimes cause an agent to try to optimize the use of time as best as it can. Using time efficiently would require an agent to decide on the optimum ordering of tasks and also be flexible enough to change the ordering as it goes along. For each task, the agent would need to decide how much time it has left and how important the task is. Perhaps some tasks are less important, making them good candidates for opportunities to make up some time. For these tasks the agent might be willing to take more risk than usual in order to possibly finish the task very quickly. For other tasks, it might be more important that they be performed on a computer that is known to handle those types of tasks correctly; a smaller risk factor would be chosen for these tasks.

Knowing the resources that are needed for each task is important, but then an agent needs to decide how important each resource is to the completion of each task. There will be many reasons for different agents to choose different importance factors for each resource. An example might be that an agent is behind schedule so it is trying to catch up a little by taking a higher risk in the server selection. If an agent is taking a higher risk, it should probably make sure that the algorithm clearly recognizes the resources that will make the risk worthwhile. The agent would need to decide on which resources will most affect the agent’s ability to perform its task quickly.

It may require lots of experience in order to make these types of decisions, but perhaps some general guidelines can be developed to help agents learn quickly.
7.10 Server Rules and Advertising

It is assumed that not all agent servers will welcome foreign agents in the same way. Some servers won’t want any foreign agents, and others will have a primary purpose of providing resources to foreign agents. There is a need for servers to inform agents as to what the rules are for use of the server.

The first problem is a matter of a server advertising itself to agents and possibly other servers. Servers need to be able to make themselves known to the agent community. Perhaps a type of “phone book” can be used in which servers can advertise. If a “phone book” idea is used, then the question of whether it should be centralized, distributed, or redundantly distributed needs to be answered. Another idea would be similar to billboards on which servers can place advertisements. No matter what solutions are developed, servers will need to have some mechanism for making themselves known to the world (or at least to those agents and servers whom it is interested in serving).

The second problem involves rules that a server would establish for agents that want to use its resources. This could be similar to globally accessible WWW pages as opposed to WWW pages that require a password for access. Taking this to another level, agent servers would probably want to specify different levels of access to various resources for different groups of agents and specific individual agents. These rules would define the permitted quantities of each resource to which particular agents are allowed access. A standard method of specifying these rules would be greatly desired. It is assumed that a server administrator would be responsible for organizing these rules, and it would be very similar to the work that system administrators are responsible for currently. Perhaps the biggest challenge here is the enforcement of such rules. If an agent is only allowed to use five percent of the available processing time, how does the system enforce this rule? Perhaps new operating systems need to be developed, or existing ones would need some revamping. In either case, this enforcement will be necessary if an agent server is going to allow unknown agents to use its resources.
7.11 Server and Agent Police

Just as with humans, there will be those servers who follow the rules and there will be those who won’t. Humans have developed the concept of police forces to keep the law-abiding people safe from those who aren’t. The same type of concept will need to be applied to agents and agent servers.

Some agents will eventually carry electronic money with them as they traverse many different and sometimes unfamiliar networks. Sensitive data, such as money, will need to be protected very carefully. Agents are not very different from humans. Agents can be attacked by other agents, and agents can be maliciously treated by the servers that they visit and the networks that they traverse. Agents can also be malicious toward the servers and/or networks that they visit.

Perhaps some form of electronic police force is necessary. This police force could wander around networks checking on agents and servers to make sure that there is no wrongdoing happening and if there is, perhaps do something about it. This proposition definitely raises some very interesting questions both technically and morally. How would an agent be certified to be a police agent? What kinds of activities are considered illegal in the agent world? How would a police agent recognize a wrongdoing and even more difficult, how would a police agent stop a malicious agent or server? What mechanisms need to be in place to make sure that the police agents don’t abuse their power? What would be a police agent’s territory? What methods should a police agent be allowed to use in order to collect information regarding other agents or servers? It is possible that some servers would try to corrupt the police agents. It is important to research methods for creating police agents with a high resistance to corruption. Practically any questions that are raised about human police forces need to be raised for agent police, and then some further questions, which are software agent specific, need to be answered.
7.12 Testing the Algorithm in Specific Environments

This algorithm was not designed with any particular environment in mind. The reason for the development of the algorithm stemmed from the realization that algorithms like this are going to be necessary as the number of uses for agents grows.

In the experiments, a simple printing environment was simulated, but perhaps the algorithm should be tested in an actual printing environment. Using the algorithm in an electronic commerce simulation would be very interesting. Any types of distributed computing that can reasonably be transformed into the mobile agent paradigm will need an algorithm that performs at least some of the functions of this algorithm. Some examples are distributed redundant file systems, distributed database systems, WWW search engines, and distributed computations.
8 Conclusion

Agents are not a new topic in the world of computers, but it has recently become very popular. A lot has been learned during the course of working on this thesis, but one point that has really been important in thinking about agents is “if the only tool you possess is a hammer, then everything looks like a nail.”[14] In other words, agents are very interesting and a popular method of software design, but they are not the solution to all software problems. OOP has its field of importance just as much as other languages have their areas of specialty such as LISP, COBOL, FORTRAN, Matlab, etc. Agents appear to have much potential for improving computer technologies, but everyone must keep in mind that agents are not the solution to all problems. Agents simply provide another tool for a designer's toolbox. Agents cannot do anything that can't be done without agents, but they can make certain tasks easier to accomplish.
Appendix A: Source Code

Appendix A.1: Algorithm Tests

```
/* File: experiment.h
* Author: Wayne Caro
* Purpose: This file provides the struct and class definitions for server and
* agent resources and servers and agents. These classes are used to simulate
* the algorithm described in the thesis, "Server Selection for Mobile Agent
* Migration" by Wayne Caro.
* The implementation files are: server.cc and agent.cc
* The main source file for this program is experiment.cc
*/
#ifndef EXPERIMENT
#define __EXPERIMENT

#define MAX_NUM_RESOURCE 25
#define MAX_RESOURCE_NAME 25
#define MAX_RESOURCE_EVALS 25
#define MAX_RESOURCES 25
#define MAX_SERVER_NAME 100
#define MAX_AGENT_NAME 100

// struct ServerResource
// Purpose: provides a method to group the information related to a resource
// located on a server
//
// struct ServerResource {
//    char name[MAX_RESOURCE_NAME];
//    int num_allowed;
//    int num_available;
//    int evals[MAX_RESOURCE_EVALS];
//    int num_evals;
//};

// struct AgentResource
// Purpose: provides a method to group the information related to a resource
// in which an agent is interested
//
// struct AgentResource {
//    char name[MAX_RESOURCE_NAME];
//    bool necessary;
//    int minimum_qty;
//    int ranking;
//};
```

82
// Class Server
// Written by: Wayne Caro
// Purpose: provides a virtual server for simulation of the agent selection
// Public functions:
// Server()
//   int Load(char *serverFileName);
//   char *getName(void) { return name; }
//   int getNumResources(void) { return num_resources; }
//   ServerResource *getResource(int index) { return &resources[index]; }
//   void Print(void);
// The layout of the server files should be :
//   <resource name> <tab> # allowed <tab> # available <evals>
// Each of the quantity values should be integer values. <evals> is a
// comma-delimited list of evaluations for each particular resource. Each
// evaluation will have a value on the range of [0..10] where the larger the
// value is, the better the evaluation.

class Server {
  public:
    // Constructor
    Server();
    // Load(): loads the server’s characteristics from "serverFileName"
    int Load(char *serverFileName);
    // getName(): returns the name of the server
    char *getName(void) { return name; }
    // getNumResources(): returns the number of resources on the server
    int getNumResources(void) { return num_resources; }
    // getResource(): returns a pointer to the ServerResource struct
    // associated with the resource at "index" location.
    ServerResource *getResource(int index) { return &resources[index]; }
    // Print(): prints the current characteristics of the server to
    // standard output
    void Print(void);
  protected:
    char name[MAX_SERVER_NAME];
    ServerResource resources[MAX_RESOURCES];
    int num_resources;
  private:
    int ParseEvals(char *str, int *evals);
};
// Class Agent
// Written by: Wayne Caro
// Purpose: allows a simulation of an agent selecting a server to which it
// would migrate.
// Public functions:
// Agent()
// int Load(char *agentFileName)
// int FindBestServer(int num_servers, Server *servers);
// void Print(void);

// The layout of the agent file should be:
// risk factor <tab> <factor>
// evaluations <tab> <number of evaluations>
// <resource name> <tab> <N/W> <tab> <minimum> <tab> <rating>
// ...
// ...
// ...

// Risk factor should be a value on the range of [0..100]. The risk factor is
// a percentage of risk that the agent is willing to accept when choosing a
// server. If the agent is willing to take 0 risk, then the server with the
// best evaluations will be selected. However, if the agent will accept 100%
// risk, then the server with the best resources will be selected. Any
// percentage inbetween 0 and 100% will cause a selection that takes both the
// resources and evaluations into account.
// The evaluations specification is an integer value that specifies up to how
// many evaluations the agent would like to have from each server for each
// resource type. This is a maximum specification, and the evaluations
// returned by the server should be the most recent ones.
// Resource names should match (case sensitively) those in the server files.
// The next field is N (necessary) or W (wanted, but not necessary). The next
// field is the minimum quantity of that resource that the agent can use.
// The last field is the rating that the agent gives the resource.
// The rating should be on the range of [0..10] (an integer value).
// The larger the rating is, the more important the resource is to the agent.
// Any empty lines will be skipped, and more than a single tab can be used for
// easier readability.

class Agent {
    public:
        // Constructor
        Agent();
        // Destructor
        ~Agent();

        // Load(): loads the characteristics of the agent from *agentFileName*
        int Load(char *agentFileName);

        // FindBestServer(): using the characteristics of the agent, this
        // algorithm will select the best server from the array "servers" and
        // return it's index value. otherwise a -1 will be returned.
        int FindBestServer(int num_servers, Server *servers);

        // Print(): prints out the current characteristics of the agent to
        // standard output
        void Print(void);

    protected:
        char name[MAX_AGENT_NAME];
        int risk_factor;
        int max_evals;

        AgentResource resources[MAX_RESOURCES];
        int num_resources;

        Server *servers;
        int num_servers;

        int *good_servers;
        int num_good_servers;
}
void FindGoodServers(void);
void WriteKmatrix(void);
void WritePmatrix(void);
void WriteAmatrix(void);
void WriteEmatrix(void);
void WriteSvector(void);

ServerResource *GetServerResource(Server *server, char *resource);
bool ServerHasResource(Server *server, AgentResource *resource);

int AverageEvals(int num_evals, int *evals);
double FindCmax(void);
int FindVectorB(double *vectorB);
int FindVectorD(double C_max, int *vectorD);

int RunAlgorithm(void);
int CrunchMatrices(void);

#endif
Program Name: experiment
Written By: Wayne Caro
Purpose: This program will be used to test my algorithm for an agent to select the best Agent Environment Server (AES) to which it should migrate.

Usage:
experiment server_file1 [server_file2... server_fileN...] agent_file

The layout of the server files should be:
<resource name> <tab> <# allowed> <tab> <# available> <evals>
.*
.*
.*
Each of the quantity values should be integer values. <evals> is a comma-delimited list of evaluations for each particular resource. Each evaluation will have a value on the range of [0..10] where the larger the value is, the better the evaluation.

The layout of the agent file should be:
* risk factor <tab> <factor>
* evaluations <tab> <number of evaluations>
* <resource name> <tab> <N/W> <tab> <minimum> <tab> <rating>
* .
* .

Risk factor should be a value on the range of [0..100]. The risk factor is a percentage of risk that the agent is willing to accept when choosing a server. If the agent is willing to take 0 risk, then the server with the best evaluations will be selected. However, if the agent will accept 100% risk, then the server with the best resources will be selected. Any percentage inbetween 0 and 100% will cause a selection that takes both the resources and evaluations into account.
The evaluations specification is an integer value that specifies up to how many evaluations the agent would like to have from each server for each resource type. This is a maximum specification, and the evaluations returned by the server should be the most recent ones.
Resource names should match (case sensitively) those in the server files.
The next field is N (necessary) or W (wanted, but not necessary). The next field is the minimum quantity of that resource that the agent can use.
The last field is the rating that the agent gives the resource.
The rating should be on the range of (0..10] (an integer value).
The larger the rating is, the more important the resource is to the agent.

Any empty lines will be skipped, and more than a single tab can be used for easier readability.
******************************************************************************/
#include <stdio.h>
#include "experiment.h"
int main(int argc, char **argv)
{
    Server *servers = NULL;
    Agent *agent = new Agent;

    if (argc < 3) {
        fprintf(stderr, "Usage: %s <server list> <agent>\n\n", argv[0]);
        exit(1);
    }

    int numServer = argc - 2; // 1 for executable; 1 for agent file
    int firstServer = 1;
    int lastServer = numServer;
    int agentIndex = argc - 1;

    servers = new Server[numServer];

    if ((agent == NULL) || (servers == NULL)) {
        fprintf(stderr, "Failed to allocate storage for servers and agent\n");
        exit(1);
    }

    for (int i=1; i <= (argc - 2); i++) {
        if (servers[i-1].Load(argv[i]) != 1)
            exit(2);
        //servers[i-1].Print();
    }

    if (agent->Load(argv[agentIndex]) != 1)
        exit(3);
    //agent->Print();

    int bestServer = agent->FindBestServer(numServer, servers);

    if (bestServer >= 0) {
        printf("The best server is: %s\n\n", servers[bestServer].getName());
    } else {
        printf("No "Best" server found\n\n");
    }

    return 0;
}
/*****************************************************************************
* File: server.cc
* Author: Wayne Caro
* Purpose: provides the implementation of class Server as defined in
* server.h.
******************************************************************************/
#include <string.h>
#include <ctype.h>
#include <stdio.h>
#include "experiment.h"

Server::Server()
{    
  num_resources = 0;
}

int Server::Load(char *fileName)
{
  FILE *fd = NULL;
  char evals[MAX_RESOURCE_EVALS * 3];   // 2 digits, plus a comma
  if ((fd = fopen(fileName, "r")) == NULL) {
    return 0;
  }

  strcpy(name, fileName);

  for (num_resources=0; !feof(fd);) {
    int num;
    int i = num_resources;
    num = fscanf(fd, "%s %d %d", resources[i].name, 
                &resources[i].num_allowed, 
                &resources[i].num_available);

    if (num == 3) {
      fgets(evals, sizeof(evals), fd);
      resources[i].num_evals = ParseEvals(evals, resources[i].evals);
      num_resources++;
    }
  }

  fclose(fd);

  return 1;
}

int Server::ParseEvals(char *str, int *evals)
{
  int count = 0;
  for (; isspace(*str) && (*str != '\0'); str++) {
    char *token = strtok(str, " ", "");
    while (token != NULL) {
      sscanf(token, "%d", &evals[count]);
      count++;
      token = strtok(NULL, ", ");
    }
  }

  return count;
}

void Server::Print(void)
{
  printf("Server Name: \n", name);
  printf("Number of resources: \n", num_resources);
  for (int i=0; i < num_resources; i++) {
    printf("\t%s\t%d\t%d\t\n", resources[i].name, 
            resources[i].num_allowed,
            resources[i].num_available,
            resources[i].num_evals);
  }
}
resources[i].num_available);

for (int j=0; j < resources[i].num_evals-1; j++) {
    printf("%d,", resources[i].evals[j]);
}
if (resources[i].num_evals > 0) {
    printf("%d", resources[i].evals[resources[i].num_evals - 1]);
}
printf("\n");
printf("\n");
/*******************************************************************************
* File: agent.cc
* Author: Wayne Caro
* Purpose: provides the implementation of class Agent as defined in
* experiment.h. This agent is used to test the algorithm proposed in the
* Master Thesis "Server Selection for Mobile Agent Migration" by Wayne Caro.
*******************************************************************************

#include <stdio.h>
#include <stdlib.h>
#include "experiment.h"
#include "matrix.h"
#include "fmatrix.h"
#define MAX_LINE 250

Agent::Agent()
{
    strcpy(name, "");
    risk_factor = 0;
    max_evals = 0;
    num_resources = 0;
}

Agent::~Agent()
{
    delete[] good_servers;
}

int Agent::Load(char *agentFileName)
{
    FILE *fd = NULL;
    char tmpStr[MAX_LINE];
    
    if ((fd = fopen(agentFileName, "r")) == NULL) {
        return 0;
    }

    strcpy(name, agentFileName);

    // read risk factor
    fscanf(fd, "%s %s %d", tmpStr, tmpStr, &risk_factor);
    fscanf(fd, "%s %d\n", tmpStr, &max_evals);

    for (num_resources=0; !feof(fd);) {
        int num;
        char necessary;
        int i = num_resources;
        num = fscanf(fd, "%s %c %d %d\n", resources[i].name,
                     &necessary,
                     &resources[i].minimum_qty,
                     &resources[i].ranking);

        if (num == 4) {
            resources[i].necessary = (necessary == 'N');
            num_resources++;
        }
    }

    fclose(fd);

    return 1;
}
int Agent::FindBestServer(int num_servers, Server *servers)
{
    this->servers = servers;
    this->num_servers = num_servers;
    num_good_servers = 0;
    good_servers = new int[num_servers];

    FindGoodServers();
    if (num_good_servers > 0) {
        WriteRmatrix();
        WritePmatrix();
        WriteAmatrix();
        WriteEmatrix();
        WriteSvector();
        return RunAlgorithm();
    }
    return -1;
}

void Agent::FindGoodServers(void)
{
    num_good_servers = 0;
    for (int i=0; i < num_servers; i++) {
        bool has_all_needed_resources = true;
        for (int j=0; has_all_needed_resources && (j < num_resources); j++) {
            if (!ServerHasResource(servers[i], resources[j]))
                if (resources[j].necessary)
                    has_all_needed_resources = false;
        }
        if (has_all_needed_resources) {
            good_servers[num_good_servers] = i;
            num_good_servers++;
        }
    }
    return;
}

void Agent::WriteRmatrix(void)
{
    FILE *fd = NULL;
    if ((fd = fopen("R.DAT", "w")) == NULL)
        return;
    for (int i=0; i < num_resources; i++) {
        for (int j=0; j < i; j++)
            fprintf(fd, "0 ");
    }
    int elem_value = 0;
    elem_value = resources[i].ranking + (resources[i].necessary?10:0);
    fprintf(fd, "%d", elem_value);
    for (int k=i + 1; k < num_resources; k++)
        fprintf(fd, "  0");
    fprintf(fd, "\n");
    fclose(fd);
}
void Agent::WritePmatrix(void)
{
    FILE *fd = NULL;
    if ((fd = fopen("P.DAT", "w")) == NULL)
        return;
    for (int i=0; i < num_resources; i++) {
        for (int j=0; j < num_good_servers; j++) {
            Server *server = &servers[good_servers[j]];
            ServerResource *server_resource;
            server_resource = GetServerResource(server, resources[i].name);
            fprintf(fd, "%d ", server_resource->num_allowed);
        }
        fprintf(fd, 
"");
    }
    fclose(fd);
}

void Agent::WriteAmatrix(void)
{
    FILE *fd = NULL;
    if ((fd = fopen("A.DAT", "w")) == NULL)
        return;
    for (int i=0; i < num_resources; i++) {
        for (int j=0; j < num_good_servers; j++) {
            Server *server = &servers[good_servers[j]];
            ServerResource *server_resource;
            server_resource = GetServerResource(server, resources[i].name);
            fprintf(fd, "%d ", server_resource->num_available);
        }
        fprintf(fd, 
"");
    }
    fclose(fd);
}

void Agent::WriteEmatrix(void)
{
    FILE *fd = NULL;
    if ((fd = fopen("E.DAT", "w")) == NULL)
        return;
    for (int i=0; i < num_resources; i++) {
        for (int j=0; j < num_good_servers; j++) {
            Server *server = &servers[good_servers[j]];
            ServerResource *server_resource;
            server_resource = GetServerResource(server, resources[i].name);
            int avg_eval = AverageEvals(server_resource->num_evals,
                                          server_resource->evals);
            fprintf(fd, "%d ", avg_eval);
        }
        fprintf(fd, 
"");
    }
    fclose(fd);
}
void Agent::WriteSvector(void)
{
    FILE *fd = NULL;
    if ((fd = fopen("S.DAT", "w")) == NULL)
        return;
    for (int i=0; i < num_resources; i++) {
        fprintf(fd, "1 ");
    }
    fprintf(fd, "\n");
    fclose(fd);
}

bool Agent::ServerHasResource(Server *server, AgentResource *resource)
{
    bool ret = false;
    ServerResource *server_resource = NULL;
    server_resource = GetServerResource(server, resource->name);
    if (server_resource != NULL) {
        if (server_resource->num_allowed >= resource->minimum_qty) {
            ret = true;
        }
    }
    return ret;
}

ServerResource *Agent::GetServerResource(Server *server, char *resource)
{
    bool ret = false;
    ServerResource *server_resource = NULL;
    for (int i=0; (ret == false) && (i < server->getNumResources()); i++) {
        server_resource = server->getResource(i);
        if (!strcmp(server_resource->name, resource)) {
            ret = true;
        }
    }
    if (!ret) server_resource = NULL;
    return server_resource;
}

int Agent::AverageEvals(int num_evals, int *evals)
{
    int total = 0;
    int count = 0;
    for (int i=0; (i < num_evals) && (i < max_evals); i++) {
        total += evals[i];
        count++;
    }
    if (count > 0) {
        total /= count;
    } else {
        total = 0;
    }
    return total;
}
int Agent::RunAlgorithm(void)
{
    CrunchMatrices();

    double C_max = FindCmax();
    if (C_max < 0) {
        return -1;
    }

    int *D = new int[num_good_servers];
    int num_in_D = 0;
    if ((num_in_D = FindVectorD(C_max, D)) < 0) {
        delete[] D;
        return -1;
    }

    double *B = new double[num_good_servers];
    int num_in_B = 0;
    if ((num_in_B = FindVectorB(B)) < 0) {
        delete[] D;
        delete[] B;
        return -1;
    }

    int best_server = -1;
    double max_value = -1.0;
    for (int i=0; i < num_in_D; i++) {
        if (i == 0) {
            best_server = good_servers[D[0]];
            max_value = B[D[0]];
        } else {
            if (B[D[i]] > max_value) {
                best_server = good_servers[D[i]];
                max_value = B[D[i]];
            }
        }
    }

    return best_server;
}

double Agent::FindCmax(void)
{
    FILE *fd = NULL;

    if ((fd = fopen("C.DAT", "r")) == NULL) {
        return -1;
    }

    double C_max = -1.0;
    for (int i=0; i < num_good_servers; i++) {
        double fTmp;
        fscanf(fd, " %lf ", &fTmp);
        if (i == 0) {
            C_max = fTmp;
        } else {
            if (fTmp > C_max) {
                C_max = fTmp;
            }
        }
    }

    printf("Cmax = %lf\n", C_max);
    fclose(fd);
    return C_max;
}
int Agent::FindVectorB(double *B)
{
    FILE *fd = NULL;
    if ((fd = fopen("B.DAT", "r")) == NULL) {
        return -1;
    }

    int num_in_B = 0;
    for (int i=0; i < num_good_servers; i++) {
        double fTmp;
        fscanf(fd, "%lf", &fTmp);
        B[num_in_B] = fTmp;
        num_in_B++;
    }
    fclose(fd);
    return num_in_B;
}

int Agent::FindVectorD(double C_max, int *D)
{
    FILE *fd = NULL;
    if ((fd = fopen("C.DAT", "r")) == NULL) {
        return -1;
    }

    int num_in_D = 0;
    for (int i=0; i < num_good_servers; i++) {
        double fTmp;
        fscanf(fd, "%lf", &fTmp);
        if (fTmp >= (C_max * (1 - (double)risk_factor/100))) {
            D[num_in_D] = i;
            num_in_D++;
        }
    }
    fclose(fd);
    return num_in_D;
}

int Agent::CrunchMatrices(void)
{
    //system("matlab < alg.m");
    FileMatrix R("R.DAT"); R.load();
    FileMatrix P("P.DAT"); P.load();
    FileMatrix A("A.DAT"); A.load();
    FileMatrix E("E.DAT"); E.load();
    FileMatrix S("S.DAT"); S.load();
    FileMatrix B("B.DAT");
    FileMatrix C("C.DAT");

    B = S*(R*(P+A));
    C = S*(R*E);

    B.save();
    C.save();
    return 1;
}
void Agent::Print(void)
{
    printf("Agent Name: %s\n", name);
    printf("Risk Factor: %d\n", risk_factor);
    printf("Max Evaluations: %d\n", max_evals);
    printf("Number of resources: %d\n", num_resources);
    for (int i=0; i < num_resources; i++) {
        printf("\t%s\t%c\t%d\t%d\n", resources[i].name,
                resources[i].necessary?‘N’:‘W’,
                resources[i].minimum_qty,
                resources[i].ranking);
    }
    printf("\n");
}
printf("\n");
Appendix A.2: Printer Simulations

/*******************************************************************************/
/*
* File: scheduler.h
* Author: Wayne Caro
* Purpose: This file provides the struct and class definitions for
* The implementation files are: scheduler.cc
* The main source files for this program are:
*  * test_change_job.cc
*  * test_matrix.cc
*  * test_one_good_printer.cc
*  * test_printer.cc
*  * test_rating.cc
*  * test_same_job.cc
* 
*******************************************************************************/
ifndef __SCHEDULER
#define __SCHEDULER

#include "printer.h"
#include "job.h"
#include "matrix.h"

#define MAX_PRINTERS 15

// struct SPrinter
// Purpose: provides a method to group the information regarding individual
// printers being simulated.
//
struct SPrinter {
    Printer *printer;
    double bScore;
    bool eligible;

    SPrinter()
    {
        printer = NULL;
        eligible = false;
    }
};

// struct CSPrinter
// Purpose: a container class for SPrinter’s
//
struct CSPrinter {
    int iNumPrinters;
    SPrinter *printers;

    CSPrinter()
    {
        iNumPrinters = 0;
        printers = NULL;
    }
};
// Class Scheduler
// Written by: Wayne Caro
// Purpose: implements the algorithm described in the Master Thesis by Wayne Caro, "Server Selection Algorithm for Mobile Agent Migration", to simulate the scheduling of print jobs on a cluster of printers.
// Public functions:
// Scheduler();
// void addPrinter(Printer *newPrinter);
// Printer *scheduleJob(Job *newJob);

class Scheduler {
public:
    // Scheduler(): constructor
    Scheduler();
    // ~Scheduler(): destructor
    ~Scheduler();
    // addPrinter(): add a new printer to the cluster and to make it available for jobs.
    void addPrinter(Printer *newPrinter);
    // schedulerJob(): ask the scheduler to try to schedule the newJob
    Printer *scheduleJob(Job *newJob);
    // getCvector(): provides a look at the internal workings of the algorithm by providing the value of the c-vector for a particular job. The job is not scheduled; it is like performing the decision process without placing the job in a printer queue.
    Matrix Scheduler::getCvector(Job *newJob);

protected:
    int m_iNumPrinters;
    Printer *m_arrPrinters[MAX_PRINTERS];

    int findValidPrinters(Job *newJob, CSPrinter *arrSPrinters);
    createRmatrix(Matrix &R, Job *newJob);
    createSvector(Matrix &s, Job *newJob);
    createAmatrix(Matrix &A, Job *newJob, CSPrinter *SPrinters);
    createEmatrix(Matrix &E, Job *newJob, CSPrinter *SPrinters);
    findEligiblePrinters(Matrix &c, Job *newJob, CSPrinter *SPrinters);
    fillInBscore(Matrix &b, CSPrinter *SPrinters);
    Printer *findBestPrinter(CSprinter *SPrinters);
    double findCmax(Matrix &c);
};

#endif
/********************************************************************
***********
* File: printer.h
* Author: Wayne Caro
* Purpose: This file provides the class definition for a printer object that
* loads its initial settings from a file.
* The implementation source file is printer.cc
*
*********************************************************************/
#ifndef __PRINTER
#define __PRINTER

#include <math.h>
#define MAX_NUM_RESOURCE 25
#define MAX_RESOURCE_NAME 25
#define MAX_RESOURCES 25
#define MAX_PRINTER_NAME 100
#define ON_TIME_EVAL    "OnTime"

// struct PrinterResource
// Written by: Wayne Caro
// Purpose: provides a simple means of grouping a printer’s resource with its
// current evaluation score.
//
struct PrinterResource {
    char name[MAX_RESOURCE_NAME];
    double eval;
};

// Class Printer
// Written by: Wayne Caro
// Purpose: provides a means of simulating a printer.
// Public functions are:
//    Printer();
//    int Load(char *printerFileName);
//    char *getName(void)
//    int getNumResources(void)
//    PrinterResource *getResource(char *name)
//    PrinterResource *getResource(int index)
//    int getDPI(void)
//    int getSpeed(void)
//    int getCurrentQueue(void)
//    double getResourceEval(char *name)
//    double getResourceEval(int index)
//    setResourceEval(char *name, double new_eval)
//    setResourceEval(int index, double new_eval)
//    void timeClick(void);
//    void newJob(long pages)
//    void Print(void);
//
The format of the data file is such that each line must contain a different setting/resource. The name of each setting/resource is followed by a tab then a number. For resources, that value is the current evaluation value on the range [0..10]. The currently recognized settings are "speed" (speed of printer in pages/min), "dpi" (dots per inch... typically 600,1200...), and "current_queue" (current number of images in the printer's queue). A particular resource that is recognized is "OnTime" which represents how well previous jobs had been printed by the desired completion time.

A sample input file is:

speed 1000
dpi 600
current_queue 15

red_paper 7
stapler 6
OnTime 4

class Printer {
public:
    // Printer(): constructor
    Printer();

    // Load(): loads the settings/features from printerFileName
    int Load(char *printerFileName);

    // getName(): returns the name of the printer
    char *getName(void)
      { return m_szName; }

    // getNumResources(): returns the number of features for the printer
    int getNumResources(void)
      { return m_iNumResources; }

    // getResource(): returns a pointer to the resource structure
    // associated with the particular name.
    PrinterResource *getResource(char *name);

    // getResource(): returns a pointer to the resource structure
    // associated with the index value.
    PrinterResource *getResource(int index)
      { return &m_arrResources[index]; }

    // getDPI(): returns the DPI of the printer
    int getDPI(void)
      { return m_iDPI; }

    // getSpeed(): returns the speed of the printer
    int getSpeed(void)
      { return m_iSpeed; }

    // getCurrentQueue(): returns the number of images currently queued
    int getCurrentQueue(void)
      { return (int)ceil(m_iCurrentQueue); }

    // getResourceEval(): returns the current evaluation value for the
    // feature that is called name.
    double getResourceEval(char *name);

    // getResourceEval(): returns the current evaluation value for the
    // feature with the index value.
    double getResourceEval(int index)
      { return m_arrResources[index].eval; }
// setResourceEval(): sets the resource, name, with the specified new
// evaluation value.
setResourceEval(char *name, double new_eval);

// setResourceEval(): sets the resource, designated by index, with
// the specified new evaluation value.
setResourceEval(int index, double new_eval)
    { m_arrResources[index].eval = new_eval; }

// timeClick(): specifies that one time click (second) has elapsed so
// that the printer object can update its counters.
void timeClick(void);

// newJob(): informs the printer that a new job of pages length has
// been added to the queue.
void newJob(long pages)
    { m_iCurrentQueue += (double)pages; }

// Print(): prints the current settings/features of the printer to
// standard output.
void Print(void);

protected:
    char m_szName[MAX_PRINTER_NAME];
    PrinterResource m_arrResources[MAX_RESOURCES];
    int m_iNumResources;
    int m_iSpeed;
    int m_iDPI;
    double m_iCurrentQueue; // number of pages currently scheduled
};

#endif
/*******************************************************************************
* File: job.h
* Author: Wayne Caro
* Purpose: This file provides the class definition for the Job class which
* will hold all the features of a print job
* The implementation source file is job.cc
*******************************************************************************
 ifndef __JOB
 define __JOB

#include "feature.h"

#define MAX_FEATURES 20

// Class Job
// Written by: Wayne Caro
// Purpose: provides a container for all the features related to a print job
// Public functions:
// Job();
// void setRisk(int risk);
// void setDPI(int DPI);
// void setMinTime(int time, bool needed, int ranking);
// void setNumPages(int pages);
// void addFeature(Feature *newFeature);
//
// int getRisk(void);
// int getDPI(void);
// int getMinTime(int *time, bool *needed, int *ranking);
// int getNumPages(void);
// int getNumFeatures(void);
// Feature *getFeature(int index);
// void Print(void);
//
 class Job {
 public:
 // Job(): constructor
 Job();

 // ~Job(): destructor
 ~Job();

 // setRisk(): sets the risk factor (0-100%)
 void setRisk(int risk);

 // setDPI(): sets the minimum desired DPI
 void setDPI(int DPI);

 // setMinTime(): sets the minimum time for the job to be completed
 // from the time that it is submitted. Other factors that must be
 // specified is whether the time is a hard-deadline (needed), and how
 // important it is to meet the specified deadline (ranking [0..10])
 void setMinTime(int time, bool needed, int ranking);

 // setNumPages(): specify the number of pages in the job
 void setNumPages(int pages);

 // addFeature(): add a new feature object to the job
 void addFeature(Feature *newFeature);

 // getRisk(): returns the specified risk factor
 int getRisk(void);
// getDPI: returns the specified minimum DPI
int getDPI(void);

// getMinTime(): returns the information related to the minimum time
// setting. The actual return value is the time.
int getMinTime(int *time, bool *needed, int *ranking);

// getNumPages(): returns the number of page of the job
int getNumPages(void);

// getNumFeatures(): returns the number of features associated with
// the job.
int getNumFeatures(void);

// getFeature(): returns the feature at "index" in the list
Feature *getFeature(int index);

// Print(): prints the current settings for the job to standard output
void Print(void);

protected:
    int m_iRisk;
    int m_iDPI;
    int m_iMinTime;
    bool m_bMinTimeNeeded;
    int m_iMinTimeRanking;
    int m_iNumPages;

    int m_iNumFeatures;
    Feature *m_arrFeatures[MAX_FEATURES];
};

#endif
ifndef FEATURE
#define FEATURE
#define MAX_FEATURE_NAME 30

class Feature {
public:
  // Feature(): basic constructor
  Feature();

  // Feature(): constructor allowing the specification of the
  // feature’s name
  Feature(char *name);

  // Feature(): constructor allowing the specification of the feature’s
  // name, whether it is needed, and a importance ranking
  Feature(char *name, bool needed, int ranking);

  void setNeeded(bool needed);
  void setRanking(double ranking);
  char *getName(void);
  bool getNeeded(void);
  double getRanking(void);

  // ~Feature(): deconstructor
  ~Feature();

  void setNeeded(bool needed);
  void setRanking(double ranking);
  char *getName(void);
  bool getNeeded(void);
  double getRanking(void);
protected:
  char m_szName[MAX_FEATURE_NAME];
  bool m_bNeeded;
  double m_fRanking;
};
#endif
/*******************************************************************************
* File: scheduler.cc
* Author: Wayne Caro
* Purpose: provides the implementation of class Scheduler as defined in
* scheduler.h.
********************************************************************************/
#include <stdlib.h>
#include <stdio.h>

// define _DEBUG
#define _DEBUG

#include "scheduler.h"
#include "matrix.h"
#include "printer.h"

Scheduler::Scheduler()
{
    m_iNumPrinters = 0;
    for (int i=0; i < MAX_PRINTERS; i++) {
        m_arrPrinters[i] = NULL;
    }
}

Scheduler::~Scheduler()
{
}

void Scheduler::addPrinter(Printer *newPrinter)
{
    m_arrPrinters[m_iNumPrinters] = newPrinter;
    m_iNumPrinters++;
}
Printer *Scheduler::scheduleJob(Job *newJob) {
    Printer *pPrinter = NULL;
    CSPrinter printersContainer;
    printersContainer.printers = new SPrinter[MAX_PRINTERS];
    printersContainer.iNumPrinters = 0;
    int numJobFeatures = newJob->getNumFeatures();

    Matrix R(numJobFeatures + 1, numJobFeatures + 1);
    createRmatrix(R, newJob);
    #ifdef _DEBUG
        R.print();
    #endif
    Matrix s(1, numJobFeatures + 1);
    createSvector(s, newJob);

    int numValidPrinters = findValidPrinters(newJob, &printersContainer);

    if (numValidPrinters > 1) {
        Matrix A(numJobFeatures + 1, numValidPrinters);
        createAmaatrix(A, newJob, &printersContainer);
        Matrix E(numJobFeatures + 1, numValidPrinters);
        createEmatrix(E, newJob, &printersContainer);

        Matrix c = s * (R * E);
        #ifdef _DEBUG
            c.print();
        #endif
        findEligiblePrinters(c, newJob, &printersContainer);

        Matrix b = s * (R * A);
        fillInBscore(b, &printersContainer);

        pPrinter = findBestPrinter(&printersContainer);
        pPrinter->newJob(newJob->getNumPages());
    } else if (numValidPrinters == 1) {
        pPrinter = printersContainer.printers[0].printer;
        pPrinter->newJob(newJob->getNumPages());
    } else {
        pPrinter = NULL;
    }

    delete[] printersContainer.printers;

    return pPrinter;
}
Matrix Scheduler::getCvector(Job *newJob)
{
    CSPrinter printersContainer;
    printersContainer.printers = new SPrinter[MAX_PRINTERS];
    printersContainer.iNumPrinters = 0;
    int numJobFeatures = newJob->getNumFeatures();
    Matrix cTmp;

    Matrix s(1, numJobFeatures + 1);
    createSvector(s, newJob);

    int numValidPrinters = findValidPrinters(newJob, &printersContainer);
    if (numValidPrinters > 0) {
        Matrix A(numJobFeatures + 1, numValidPrinters);
        createAmatrix(A, newJob, &printersContainer);
        Matrix E(numJobFeatures + 1, numValidPrinters);
        createEmatrix(E, newJob, &printersContainer);

        cTmp = s * E;
    }
    delete[] printersContainer.printers;

    return cTmp;
}
int Scheduler::findValidPrinters(Job *newJob, CSPrinter *SPrinters)
{
    int numPrinters = 0;
    SPrinters->iNumPrinters = 0;

    for (int j=0; j < m_iNumPrinters; j++) {
        bool bGotAllNeeded = true;
        Printer *printer = m_arrPrinters[j];

        // check all features
        for (int i=0; (i < newJob->getNumFeatures()) && bGotAllNeeded; i++) {
            char *featureName = newJob->getFeature(i)->getName();
            if (printer->getResource(featureName) == NULL) {
                bGotAllNeeded = false;
            }
        }

        // check for print time
        int time, ranking;
        bool needed;
        newJob->getMinTime(&time, &needed, &ranking);
        if (needed) {
            int totalPages = newJob->getNumPages() + printer->getCurrentQueue();
            int timeWillTake = totalPages / printer->getSpeed();
            if (timeWillTake > time) {
                bGotAllNeeded = false;
            }
        }

        // check DPI
        if (newJob->getDPI() > printer->getDPI()) {
            bGotAllNeeded = false;
        }

        if (bGotAllNeeded) {
            SPrinters->printers[numPrinters].printer = printer;
            numPrinters++;
            #ifdef _DEBUG
            printf("Printer %s is valid\n", printer->getName());
            #endif
        }
    }  

    SPrinters->iNumPrinters = numPrinters;

    return SPrinters->iNumPrinters;
}
Scheduler::createRmatrix(Matrix &R, Job *newJob)
{
    // fill in diagonal for features...
    //
    for (int i=0; i < newJob->getNumFeatures(); i++) {
        // fill in the leading 0's
        for (int j=0; j < i; j++) {
            R.set(i, j, 0.0);
        }
        // diagonal value is the ranking plus 10: if needed, 0: if not needed
        //
        double elem_value = (double)newJob->getFeature(i)->getRanking();
        elem_value += (newJob->getFeature(i)->getNeeded() ? 10.0 : 0.0);
        R.set(i, i, elem_value);
        //
        for (int k=i + 1; k < newJob->getNumFeatures(); k++) {
            R.set(i, k, 0.0);
        }
    }

    // fill in bottom right corner with value for print time
    int time, ranking;
    newJob->getMinTime(&time, &needed, &ranking);
    double elem_value = (double)ranking;
    elem_value += (needed ? 10.0 : 0.0);
    R.set(newJob->getNumFeatures(), newJob->getNumFeatures(), elem_value);
}

Scheduler::createSvector(Matrix &s, Job *newJob)
{
    for (int i=0; i < newJob->getNumFeatures() + 1; i++) {
        s.set(0, i, 1.0);
    }
}

Scheduler::createAmatrix(Matrix &A, Job *newJob, CSPrinter *SPrinters)
{
    double fTmp = 1.0;
    for (int i=0; i < newJob->getNumFeatures(); i++) {
        for (int j=0; j < SPrinters->iNumPrinters; j++) {
            A.set(i, j, fTmp);
        }
    }

    // The score for the print speeds is (x*s)/(x_0 + x)
    //
    //   x : new job size (in pages)
    //   x_0 : current number of pages queued on printer
    //   s : speed of printer (pages/time_click)
    //
    int row = newJob->getNumFeatures();
    for (int j=0; j < SPrinters->iNumPrinters; j++) {
        //
        SPrinter *pTmp = &SPrinters->printers[j];
        Printer *printer = pTmp->printer;

        fTmp = (double)printer->getSpeed();
        fTmp *= newJob->getNumPages();
        fTmp /= ((double)printer->getCurrentQueue() + newJob->getNumPages());
        A.set(row, j, fTmp);
    }
}
Scheduler::createEmatrix(Matrix &E, Job *newJob, CSPrinter *SPrinters) {
    for (int i=0; i < newJob->getNumFeatures(); i++) {
        for (int j=0; j < SPrinters->iNumPrinters; j++) {
            double fTmp;
            char *featureName = newJob->getFeature(i)->getName();
            SPrinter *pTmp = &SPrinters->printers[j];
            fTmp = (double)pTmp->getResourceEval(featureName);
            E.set(i, j, fTmp);
        }
    }

    int row = newJob->getNumFeatures();
    for (int j=0; j < SPrinters->iNumPrinters; j++) {
        double fTmp;
        SPrinter *pTrnp = &SPrinters->printers[j];
        fTmp = (double)pTrnp->printer->getResourceEval(ON_TIME_EVAL);
        E.set(row, j, fTmp);
    }
}

Scheduler::findEligiblePrinters(Matrix &c, Job *newJob, CSPrinter *SPrinters) {
    int col_in_C = 0;
    double C_max = findCmax(c);
    double risk_factor = (double)newJob->getRisk();

    for (int i=0; i < SPrinters->iNumPrinters; i++) {
        double fTmp = c.get(0, col_in_C);
        if (fTmp >= (C_max * (1 - risk_factor/100)) ) {
            SPrinters->printers[i].eligible = true;
            #ifdef _DEBUG
                Printer *printer = SPrinters->printers[i].printer;
                printf("Printer %s is eligible\n", printer->getName());
            #endif
        } else {
            SPrinters->printers[i].eligible = false;
            #ifdef _DEBUG
                Printer *printer = SPrinters->printers[i].printer;
                printf("Printer %s is NOT eligible\n", printer->getName());
            #endif
        }
        col_in_C++;
    }
}

Scheduler::fillInBsore(Matrix &b, CSPrinter *SPrinters) {
    for (int i=0; i < SPrinters->iNumPrinters; i++) {
        SPrinters->printers[i].bScore = b.get(0, i);
    }
}
Printer *Scheduler::findBestPrinter(CSPrinter *SPrinters)  
{  
    SPrinter *best = NULL;  
    bool found = false;  
    for (int i=0; i < SPrinters->iNumPrinters; i++) {  
        #ifdef _DEBUG  
            Printer *printer = SPrinters->printers[i].printer;  
            printf("Printer %s has bscore %8.2lf\n", printer->getName(),  
                SPrinters->printers[i].bScore);  
        #endif  
        if (!found && SPrinters->printers[i].eligible) {  
            best = &SPrinters->printers[i];  
            found = true;  
        } else if (SPrinters->printers[i].eligible) {  
            if (SPrinters->printers[i].bScore > best->bScore) {  
                best = &SPrinters->printers[i];  
            }  
        }  
    }  
    return best->printer;  
}  

double Scheduler::findCmax(Matrix &c)  
{  
    double Cmax = c.get(0, 0);  
    for (int i=0; i < c.get_columns(); i++) {  
        if (c.get(0, i) > Cmax) {  
            Cmax = c.get(0, i);  
        }  
    }  
    return Cmax;  
}
/* File: printer.cc 
* Author: Wayne Caro 
* Purpose: provides the implementation of class Printer as defined in 
* printer.h. 
* */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include "printer.h" 

Printer::Printer()
{
    strcpy(m_szName, "");
    m_iNumResources = 0;
}

int Printer::Load(char *printerFileName)
{
    int ret = 1;
    FILE *fd = NULL;
    char szTmp[100];
    int iTmp;
    
    strcpy(m_szName, printerFileName);
    
    if ((fd = fopen(m_szName, "r")) == NULL) {
        fprintf(stderr, "Failed to load printer: %s\n", m_szName);
        return 0;
    }
    m_iNumResources = 0;
    while (!feof(fd)) {
        if (fscanf(fd, "%s %d", szTmp, &iTmp) != 2)
            continue;
        if (!strcmp(szTmp, "speed")) m_iSpeed = iTmp;
        else if (!strcmp(szTmp, "current_queue")) m_iCurrentQueue = (double)iTmp;
        else if (!strcmp(szTmp, "dpi")) m_iDPI = iTmp;
        else {
            strcpy(m_arrResources[m_iNumResources].name, szTmp);
            m_arrResources[m_iNumResources].eval = (double)iTmp;
            m_iNumResources++;
        }
    }
    fclose(fd);
    return ret;
}
PrinterResource *Printer::getResource(char *name)  
{  
  PrinterResource *pResource = NULL;  
  for (int i=0; i < m_iNumResources; i++) {  
    if (!strcmp(m_arrResources[i].name, name)) {  
      pResource = &m_arrResources[i];  
    }  
  }  
  return pResource;  
}  

double Printer::getResourceEval(char *name)  
{  
  double ret = -1.0;  
  PrinterResource *pResource = getResource(name);  
  if (pResource != NULL) {  
    ret = pResource->eval;  
  }  
  return ret;  
}  

Printer::setResourceEval(char *name, double new_eval)  
{  
  PrinterResource *pResource = getResource(name);  
  if (pResource != NULL) {  
    pResource->eval = new_eval;  
  }  
}  

void Printer::timeClick(void)  
{  
  if (((double)m_iSpeed / 60.0) <= m_iCurrentQueue) {  
    m_iCurrentQueue -= ((double)m_iSpeed / 60.0);  
  } else {  
    m_iCurrentQueue = 0.0;  
  }  
}  

void Printer::Print(void)  
{  
  printf("Printer: \%s\n", m_szName);  
  printf("\tSpeed: %d\n", m_iSpeed);  
  printf("\tCurrently Queued: %d\n", (int)ceil(m_iCurrentQueue));  
  printf("\tDPI : %d\n", m_iDPI);  
  for (int i=0; i < m_iNumResources; i++) {  
    printf("\t%s %8.2lf\n", m_arrResources[i].name, m_arrResources[i].eval);  
  }  
}
/*******************************************************************************
* File: job.cc
* Author: Wayne Caro
* Purpose: provides the implementation of class Job as defined in
* job.h.
********************************************************************************
#include<stdio.h>
#include "job.h"

Job::Job()
{
    m_iRisk = 0;
    m_iDPI = 0;
    m_iMinTime = 0;
    m_bMinTimeNeeded = false;
    m_iMinTimeRanking = 0;
    m_iNumPages = 0;
    m_iNumFeatures = 0;
}

Job::~Job()
{
}

void Job::setRisk(int risk)
{
    m_iRisk = risk;
}

void Job::setDPI(int DPI)
{
    m_iDPI = DPI;
}

void Job::setMinTime(int time, bool needed, int ranking)
{
    m_iMinTime = time;
    m_bMinTimeNeeded = needed;
    m_iMinTimeRanking = ranking;
}

void Job::setNumPages(int pages)
{
    m_iNumPages = pages;
}

void Job::addFeature(Feature *newFeature)
{
    m_arrFeatures[m_iNumFeatures++] = newFeature;
}

int Job::getRisk(void)
{
    return m_iRisk;
}

int Job::getDPI(void)
{
    return m_iDPI;
}
int Job::getMinTime(int *time, bool *needed, int *ranking)
{
    *time = m_iMinTime;
    *needed = m_bMinTimeNeeded;
    *ranking = m_iMinTimeRanking;

    return m_iMinTime;
}

int Job::getNumPages(void)
{
    return m_iNumPages;
}

int Job::getNumFeatures(void)
{
    return m_iNumFeatures;
}

Feature *Job::getFeature(int index)
{
    return m_arrFeatures[index];
}

void Job::Print(void)
{
    printf("Job:\n");
    printf("\trisk: %d\n", m_iRisk);
    printf("\tDPI: %d\n", m_iDPI);
    printf("\tNumPages: %d\n", m_iNumPages);
    printf("\tMinTime: %d", m_iMinTime);
    printf("\t%s", m_bMinTimeNeeded?"true":"false");
    printf("\t%d\n", m_iMinTimeRanking);
    for (int i=0; i < m_iNumFeatures; i++) {
        printf("\t%s", m_arrFeatures[i]->getName());
        printf("\t%s", m_arrFeatures[i]->getNeeded()?"true":"false");
        printf("\t%d\n", m_arrFeatures[i]->getRanking());
    }
    printf("\n");
}
#include <string.h>

Feature::Feature()
{
    strcpy(m_szName, "")
    m_bNeeded = false;
    m_fRanking = 0.0;
}

Feature::Feature(char *name)
{
    strcpy(m_szName, name);
    m_bNeeded = false;
    m_fRanking = 0.0;
}

Feature::Feature(char *name, bool needed, double ranking)
{
    strcpy(m_szName, name);
    m_bNeeded = needed;
    m_fRanking = ranking;
}

Feature::~Feature()
{
}

void Feature::setNeeded(bool needed)
{
    m_bNeeded = needed;
}

void Feature::setRanking(double ranking)
{
    m_fRanking = ranking;
}

char *Feature::getName(void)
{
    return m_szName;
}

bool Feature::getNeeded(void)
{
    return m_bNeeded;
}

double Feature::getRanking(void)
{
    return m_fRanking;
}
/******************************************************************************
* File: test_one_good_printer.cc
* Author: Wayne Caro
* Usage: test_one_good_printer printer1_file printer2_file ...
* The file specifications are in printer.h
* Purpose: provides the main function for a simulation of the algorithm in the
* Master Thesis "Server Selection for Mobile Agent Migration" by Wayne Caro.
* This particular simulation uses print jobs with 2 features (duplex, and
* stapling). Only NUM_JOBS_TO_SUBMIT jobs are submitted to the scheduler.
* After each jobs has been submitted a small amount of time is ellapsed (about
* 45 seconds) to give the printers a small amount of time to print.
*******************************************************************************/
#include<stdlib.h>
#include<stdio.h>
#include "scheduler.h"
#include "job.h"
#include "printer.h"
#define NUM_JOBS_TO_SUBMIT 15

int main(int argc, char **argv)
{
    Scheduler scheduler;
    Printer *printers;
    Job tmpJob;

    printers = new Printer[argc-1];

    for (int i=1; i < argc; i++) {
        printers[i-1].Load(argv[i]);
        //printers[i-1].Print();
        scheduler.addPrinter(&printers[i-1]);
    }

    Feature duplex("duplex", true, 8);
    Feature staple("staple", true, 8);

    tmpJob.setRisk(10);
    tmpJob.setDPI(600);
    tmpJob.setMinTime(10, false, 2);
    tmpJob.setNumPages(100);
    tmpJob.addFeature(&duplex);
    tmpJob.addFeature(&staple);
printf("Printer 1 Queue\tPrinter 2 Queue\n");

for (int i=0; i < NUM_JOBS_TO_SUBMIT; i++) {
    for (int j=0; j < 45; j++) {
        for (int i=1; i < argc; i++) {
            if (j % 5 == 0) {
                printf("%d\t", printers[i-1].getCurrentQueue());
            }
            printers[i-1].timeClick();
        }
        if (j % 5 == 0) {
            printf("\n");
        }
    }
    Printer *pPrinter = scheduler.scheduleJob(&tmpJob);
    if (pPrinter == NULL) {
        printf("Couldn’t schedule job\n");
    }
}

for (int j=0; j < 45; j++) {
    for (int i=1; i < argc; i++) {
        if (j % 5 == 0) {
            printf("%d\t", printers[i-1].getCurrentQueue());
        }
        printers[i-1].timeClick();
    }
    if (j % 5 == 0) {
        printf("\n");
    }
}

return 0;
/*******************************************************************************
* File: test_same_job.cc
* Author: Wayne Caro
* Usage: test_same_job printer1_file printer2_file ...
*     The file specifications are in printer.h
* Purpose: provides the main function for a simulation of the algorithm in the
* Master Thesis "Server Selection for Mobile Agent Migration" by Wayne Caro.
* This particular simulation uses print jobs with 2 features (duplex and
* stapling). This simulation submits consecutive jobs in which the printers’
* states (features/evals), and the job features remain the same for all
* NUM_JOBS_TO_SUBMIT jobs.
********************************************************************************/
#include <stdlib.h>
#include <stdio.h>
#include "scheduler.h"
#include "job.h"
#include "printer.h"

#define NUM_JOBS_TO_SUBMIT 35

int main(int argc, char **argv)
{
    Scheduler scheduler;
    Printer *printers;
    Job tmpJob;

    printers = new Printer[argc-1];

    for (int i=1; i < argc; i++) {
        printers[i-1].Load(argv[i]);
        //printers[i-1].Print();
        scheduler.addPrinter(&printers[i-1]);
    }

    Feature duplex("duplex", true, 8);
    Feature staple("staple", true, 8);

    tmpJob.setRisk(10);
    tmpJob.setDPI(600);
    tmpJob.setMinTime(10, false, 2);
    tmpJob.setNumPages(100);
    tmpJob.addFeature(&duplex);
    tmpJob.addFeature(&staple);
printf("Printer 1 Queue\tPrinter 2 Queue\n");

for (int i=0; i < NUM_JOBS_TO_SUBMIT; i++) {
    for (int j=0; j < 45; j++) {
        for (int i=1; i < argc; i++) {
            if (j % 10 == 0) {
                printf("%d\t", printers[i-1].getCurrentQueue());
            }
            printers[i-1].timeClick();
        }
        if (j % 10 == 0) {
            printf("\n");
        }
    }
    Printer *pPrinter = scheduler.scheduleJob(&tmpJob);
    if (pPrinter == NULL) {
        printf("Couldn’t schedule job\n");
    }
}

for (int j=0; j < 45; j++) {
    for (int i=1; i < argc; i++) {
        if (j % 10 == 0) {
            printf("%d\t", printers[i-1].getCurrentQueue());
        }
        printers[i-1].timeClick();
    }
    if (j % 10 == 0) {
        printf("\n");
    }
}

return 0;
/*******************************************************************************
*
* File: test_rating.cc
* Author: Wayne Caro
* Usage: test_rating printer1_file printer2_file ..
*     The file specifications are in printer.h
* Purpose: provides the main function for a simulation of the algorithm in the
* Master Thesis "Server Selection for Mobile Agent Migration" by Wayne Caro.
* This particular simulation uses print jobs with 2 features (duplex, staple)
* This simulation submits consecutive jobs in which
* the evaluation scores for the first two printers are slowly changed. After
* BUFFER_NUM jobs, the the evals for the first printner is decremented by
* DECREMENT after each job until the number of jobs reaches
* (NUM_JOBS_TO_SUBMIT / DIV). At that point, the second printer’s evals begin
* to be slowly decremented by DECREMENT until
* there have been (NUM_JOBS_TO_SUBMIT - BUFFER_NUM) jobs submitted. Then all
* factors are kept constant until all NUM_JOBS_TO_SUBMIT have been submitted.
********************************************************************************/

#include<stdlib.h>
#include<stdio.h>
#include "scheduler.h"
#include "job.h"
#include "printer.h"

#define NUM_JOBS_TO_SUBMIT 1000
#define BUFFER_NUM 30
#define DECREMENT 0.02
#define DIV 2.5

int main(int argc, char **argv)
{
    Scheduler scheduler;
    Printer *printers;
    Job tmpJob;

    argc--;
    printers = new Printer[argc-1];

    for (int i=1; i < argc; i++) {
        printers[i-1].Load(argv[i]);
        //printers[i-1].Print();
        scheduler.addPrinter(&printers[i-1]);
    }

    Feature duplex("duplex", true, 8);
    Feature staple("staple", true, 8);

    tmpJob.setRisk(atoi(argv[argc]));
    tmpJob.setDPI(600);
    tmpJob.setMinTime(10, false, 2);
    tmpJob.setNumPages(100);
    tmpJob.addFeature(&duplex);
    tmpJob.addFeature(&staple);

    //tmpJob.Print();
printf("Printer 1 Eval\tPrinter 2 Eval\t")
printf("Printer 1 Pages\tPrinter 2 Pages\n");

for (int i=0; i < NUM_JOBS_TO_SUBMIT; i++) {
    for (int j=0; j < 45; j++) {
        if (j % 10 == 0) {
            Matrix c = scheduler.getCvector(&tmpJob);
            for (int k=0; k < c.get_columns(); k++) {
                printf("%lf\t", c.get(0, k));
            }
        }
        else {  
            printf("%d\n", printers[i-1].getCurrentQueue());
        }
    }
}
    printers[i-1].timeClick();
}

Printer *pPrinter = scheduler.scheduleJob(&tmpJob);
if (pPrinter == NULL) {
    printf("Couldn't schedule job\n");
}

if ((i > BUFFER_NUM) && (i < NUM_JOBS_TO_SUBMIT/DIV)) {
    double fTmp = printers[0].getResourceEval(0);
    if ((fTmp - DECREMENT) >= 4.0) {
        printers[0].setResourceEval(0, fTmp - DECREMENT);
    }  
    fTmp = printers[0].getResourceEval(1);
    if ((fTmp - DECREMENT) >= 4.0) {
        printers[0].setResourceEval(1, fTmp - DECREMENT);
    }
}

if ((i > NUM_JOBS_TO_SUBMIT/DIV) && (i < NUM_JOBS_TO_SUBMIT-DIV)) {
    double fTmp = printers[1].getResourceEval(0);
    if ((fTmp - DECREMENT) >= 0.0) {
        printers[1].setResourceEval(0, fTmp - DECREMENT);
    }  
    fTmp = printers[1].getResourceEval(1);
    if ((fTmp - DECREMENT) >= 0.0) {
        printers[1].setResourceEval(1, fTmp - DECREMENT);
    }
    fTmp = printers[1].getResourceEval(2);
    if ((fTmp - DECREMENT/2.8) >= 0.0) {
        printers[1].setResourceEval(2, fTmp - DECREMENT/2.8);
    }
}

return 0;}
/*************************************************************************************
* File: test_change_job.cc
* Author: Wayne Caro
* Usage: test_change_job printer1_file printer2_file ...
* The file specifications are in printer.h
* Purpose: provides the main function for a simulation of the algorithm in the
* Master Thesis "Server Selection for Mobile Agent Migration" by Wayne Caro.
* This particular simulation uses print jobs with 3 features (3-holed paper,
* red paper, and stapling). This simulation submits consecutive jobs in which
* the importance factors for the 3-holed, and red papers are slowly changed.
* First, the importance factors begin at: 3-holed=10, red=5, staple=8. After
* BUFFER_NUM jobs, the the importance factor for the 3-holed is decremented by
* DECREMENT after each job until the number of jobs reaches
* (NUM_JOBS_TO_SUBMIT / DIV). At that point, 3-holed is reset to 5 and red is
* reset to 10. Then red's factor is slowly decremented by DECREMENT until
* there have been (NUM_JOBS_TO_SUBMIT - BUFFER_NUM) jobs submitted. Then all
* factors are kept constant until all NUM_JOBS_TO_SUBMIT have been submitted.
**************************************************************************************

#include<stdlib.h>
#include<stdio.h>
#include "scheduler.h"
#include "job.h"
#include "printer.h"

#define NUM_JOBS_TO_SUBMIT 1000
#define BUFFER_NUM 30
#define DECREMENT 0.03
#define DIV 2

int main(int argc, char **argv)
{
    Scheduler scheduler;
    Printer *printers;
    Job tmpJob;

    argc--;
    printers = new Printer[argc-1];

    for (int i=1; i < argc; i++) {
        printers[i-1].Load(argv[i]);
        //printers[i-1].Print();
        scheduler.addPrinter(&printers[i-1]);
    }

    Feature holed("3-holed", false, 10);
    Feature red("red", false, 5);
    Feature staple("staple", true, 8);

    tmpJob.setRisk(atoi(argv[argc]));
    tmpJob.setDPI(600);
    tmpJob.setMinTime(10, false, 2);
    tmpJob.setNumPages(10);
    tmpJob.addFeature(&holed);
    tmpJob.addFeature(&red);
    tmpJob.addFeature(&staple);

    //tmpJob.Print();
}
printf("Red Rating\t3-Holed Rating\t");
printf("Printer 1 Pages\tPrinter 2 Pages\n");

for (int i=0; i < NUM_JOBS_TO_SUBMIT; i++) {
    for (int j=0; j < 5; j++) {
        for (int i=1; i < argc; i++) {
            printers[i-1].timeClick();
        }
    }
    Printer *pPrinter = scheduler.scheduleJob(&tmpJob);
    printf("%lf\t%lf\n", red.getRanking(), holed.getRanking());
    for (int j=1; j < argc; j++) {
        if (j == argc-1) {
            printf("%d
", printers[j-1].getCurrentQueue());
        } else {
            printf("%d\t", printers[j-1].getCurrentQueue());
        }
    }
}

//printf("New Job #\%d\n", i);
if (pPrinter == NULL) {
    printf("Couldn’t schedule job\n");
}

if ((i > BUFFER_NUM) && (i < NUM_JOBS_TO_SUBMIT/4)) {
    if (holed.getRanking() - DECREMENT > 0.0) {
        holed.setRanking(holed.getRanking() - DECREMENT);
    }
}

if (i == NUM_JOBS_TO_SUBMIT / 4) {
    holed.setRanking(5.0);
    red.setRanking(10.0);
    //printf("Resetting rankings\n");
}

if ((i > NUM_JOBS_TO_SUBMIT/4) && (i < NUM_JOBS_TO_SUBMIT-BUFFER_NUM)) {
    if (red.getRanking() - DECREMENT > 0.0) {
        red.setRanking(red.getRanking() - DECREMENT);
    }
}

return 0;
Appendix A.3: Support Files

/*******************************************************************************
*                                                                             *
* File: matrix.h                                                             *
* Author: Wayne Caro                                                          *
* Purpose: This file provides the class definition for a matrix object       *
* The implementation source file is matrix.cc                                *
*                                                                             */
ifndef MATRIX
#define MATRIX
//===
// Class Matrix
// Written by: Wayne Caro
// Purpose: provides a matrix class on which simple operations can be
// performed. The values stored in the matrix are double's.
// Public Funtions:
//   Matrix(int r, int c);
//   Matrix(void);
//   Matrix(const Matrix &matrix);
//   void zero(void);
//   void set(int row, int column, double value);
//   double get(int row, int column);
//   void print(void);
//   Matrix& operator=(const Matrix m);
//   friend Matrix operator+(const Matrix&, const Matrix&);
//   friend Matrix operator*(const Matrix&, const Matrix&);
//   int get_rows(void) { return m_iRows; }
//   int get_columns(void) { return m_iColumns; }
//===
class Matrix {
  public:
    // Matrix(): constructor allowing size specified
    Matrix(int r, int c);
    // Matrix(): constructor...size is empty.
    Matrix(void);

    // Matrix(): copy constructor
    Matrix(const Matrix &matrix);

    // ~Matrix(): destructor
    ~Matrix();

    // zero(): zero's out the entire matrix
    void zero(void);

    // set(): sets the value at row,column to be value
    void set(int row, int column, double value);

    // get(): gets the value from row,column
    double get(int row, int column);

    // print(): prints the contents of the matrix to stdout
    void print(void);

    // operator=(): copys the contents of m to itself (performs necessary
    // resizing)
    Matrix& operator=(const Matrix m);
}
friend Matrix operator+(const Matrix&, const Matrix&);

friend Matrix operator*(const Matrix&, const Matrix&);

int get_rows(void) { return m_iRows; }

int get_columns(void) { return m_iColumns; }

protected:
void allocate(int r, int c);
void deallocate(void);

int m_iRows, m_iColumns;
double **m_arrM;
};
/***************************************************************************************
* File: fmatrix.h
* Author: Wayne Caro
* Purpose: This file provides the class definition for a file-based matrix
* The implementation source file is fmatrix.cc
* ***************************************************************************************/
#ifndef __FMATRIX
#define __FMATRIX

#include "matrix.h"

#define MAX_FILENAME 100

class FileMatrix : public Matrix {
public:
    // FileMatrix(): constructor allowing specification of filename
    FileMatrix(const char *filename);
    // FileMatrix(const FileMatrix &matrix);
    void load(void);
    void save(void);
    FileMatrix& operator=(const Matrix &m);

    // Format of the matrix files is space/tab delimited between columns and
    // newline delimited between rows. Each element is stored as a
    // double-precision floating point number
    FileMatrix(const char *filename);
    FileMatrix(const FileMatrix &matrix);
    ~FileMatrix();
    void load(void);
    void save(void);
    FileMatrix& operator=(const Matrix &m);

    protected:
    int fillInArray(char *str, double *arr);
    char m_szfilename[MAX_FILENAME];
};
#endif
/*******************************************************************************
* File: matrix.cc
* Author: Wayne Caro
* Purpose: provides the implementation of class Matrix as defined in
* matrix.h.
*******************************************************************************
#include <stdio.h>
#include <stdlib.h>
#include "matrix.h"

Matrix::Matrix(int r, int c) {
    Matrix();
    allocate(r, c);
}

Matrix::Matrix(void) {
    m_iRows = m_iColumns = 0;
    m_arrM = NULL;
}

Matrix::Matrix(const Matrix &matrix) {
    allocate(matrix.m_iRows, matrix.m_iColumns);
    for (int i=0; i < m_iRows; i++) {
        for (int j=0; j < m_iColumns; j++) {
            m_arrM[i][j] = matrix.m_arrM[i][j];
        }
    }
}

Matrix::~Matrix() {
    deallocate();
}

void Matrix::zero(void) {
    for (int i=0; i < m_iRows; i++) {
        for (int j=0; j < m_iColumns; j++) {
            m_arrM[i][j] = 0.0;
        }
    }
}

void Matrix::set(int row, int column, double value) {
    m_arrM[row][column] = value;
}

double Matrix::get(int row, int column) {
    return m_arrM[row][column];
}
void Matrix::print(void) {
    for (int i=0; i < m_iRows; i++) {
        for (int j=0; j < m_iColumns; j++) {
            printf("%12.21f", m_arrM[i][j]);
        }
        printf("\n");
    }
    printf("\n");
}

void Matrix::allocate(int r, int c) {
    m_arrM = new double *[r];
    for (int i=0; i < r; i++) {
        m_arrM[i] = new double[c];
    }
    m_iRows = r;
    m_iColumns = c;
}

void Matrix::deallocate(void) {
    if (m_iRows != 0) {
        for (int i=0; i < m_iRows; i++) {
            if (m_arrM[i] != NULL) {
                delete[] m_arrM[i];
            }
        }
        delete[] m_arrM;
        m_arrM = NULL;
    }
}

Matrix& Matrix::operator=(const Matrix m) {
    deallocate();
    allocate(m.m_iRows, m.m_iColumns);
    for (int i=0; i < m_iRows; i++) {
        for (int j=0; j < m_iColumns; j++) {
            m_arrM[i][j] = m.m_arrM[i][j];
        }
    }
    return *this;
}

Matrix operator+(const Matrix &m1, const Matrix &m2) {
    Matrix Mtmp(m1.m_iRows, m1.m_iColumns);
    for (int i=0; i < Mtmp.m_iRows; i++) {
        for (int j=0; j < Mtmp.m_iColumns; j++) {
            Mtmp.m_arrM[i][j] = m1.m_arrM[i][j] + m2.m_arrM[i][j];
        }
    }
    return Mtmp;
}
Matrix operator*(const Matrix &ml, const Matrix &m2)
{
    Matrix Mtmp(ml.m_iRows, m2.m_iColumns);
    for (int i=0; i < ml.m_iRows; i++) {
        for (int j=0; j < m2.m_iColumns; j++) {
            double dTmp = 0;
            for (int k=0; k < ml.m_iColumns; k++) {
                dTmp += (ml.m_arrM[i][k] * m2.m_arrM[k][j]);
            }
            Mtmp.m_arrM[i][j] = dTmp;
        }
    }
    return Mtmp;
}
/********************************************
* File: fmatrix.cc
* Author: Wayne Caro
* Purpose: provides the implementation of class FileMatrix as defined in
* fmatrix.h.
*
*********************************************/
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#include "matrix.h"
#include "fmatrix.h"

FileMatrix::FileMatrix(const char *filename)
{
    Matrix::Matrix();
    strcpy(m_szfilename, filename);
}

FileMatrix::FileMatrix(const FileMatrix &matrix)
{
    Matrix(matrix.m_iRows, matrix.m_iColumns);
    for (int i=0; i < m_iRows; i++) {
        for (int j=0; j < m_iColumns; j++) {
            m_arrM[i][j] = matrix.m_arrM[i][j];
        }
    }
}

FileMatrix::~FileMatrix()
{
}
void FileMatrix::load(void)
{
    FILE *file;
    char szTmp[500];
    double darrTmp[100];

    if ((file = fopen(m_szfilename, "r")) == NULL) {
        fprintf(stderr, "Failed to open \"%s\" in FileMatrix::load()\n",
                m_szfilename);
        return;
    }

    fgets(szTmp, sizeof(szTmp), file);
    szTmp[strlen(szTmp) - 1] = '\0';

    m_iColumns = fillInArray(szTmp, darrTmp);

    m_arrM = new double *[200];
    for (int i=0; i < 200; i++) m_arrM[i] = NULL;

    m_iRows = 0;
    do {
        m_arrM[m_iRows] = new double[m_iColumns];
        if (m_iRows == 0) {
            memcpy(m_arrM[m_iRows], darrTmp, sizeof(double)*m_iColumns);
        } else {
            fillInArray(szTmp, m_arrM[m_iRows]);
        }

        fgets(szTmp, sizeof(szTmp), file);
        szTmp[strlen(szTmp) - 1] = '\0';
        m_iRows++;
    } while (!feof(file));

    fclose(file);
}

void FileMatrix::save(void)
{
    FILE *file;

    if ((file = fopen(m_szfilename, "w")) == NULL) {
        fprintf(stderr, "Failed to open file in FileMatrix::save()\n");
        return;
    }

    for (int i=0; i < m_iRows; i++) {
        for (int j=0; j < m_iColumns; j++) {
            fprintf(file, "%12.21f", m_arrM[i][j]);
        }
        fprintf(file, "\n");
    }

    fclose(file);
}
FileMatrix& FileMatrix::operator=(const Matrix &m)
{
    Matrix::operator=(m);
    return *this;
}

int FileMatrix::fillInArray(char *str, double *arr)
{
    int count = 0;
    char *token = strtok(str, " \	");

    while (token != NULL) {
        sscanf(token, " %lf ", &arr[count]);
        count++;
        token = strtok(NULL, " \	");
    }
    return count;
}
## Appendix B: Data Files

The following table provides the location (on the CDR) of the data files for the algorithm tests that were explained in section 6.1.

<table>
<thead>
<tr>
<th>Test</th>
<th>Location (base directory: \Experimentation\alg_tests)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test #1: Only One Server Has All the Necessary Resources</td>
<td>one_all_need\</td>
</tr>
<tr>
<td>Test #2: One Server Has Much Better Resources Availability</td>
<td>one_better_avail\</td>
</tr>
<tr>
<td>Test #3: One Server Has Much Better Resource Evaluations</td>
<td>one_better_evals\</td>
</tr>
<tr>
<td>Test #4: One Server Permits Much Greater Quantity of Resources</td>
<td>one_more_perm\</td>
</tr>
<tr>
<td>Test #5: Best Resources, but Lowest Evaluations</td>
<td>one_better_resource_lowest_evals\</td>
</tr>
<tr>
<td>Test #6: Perfect Evaluations, Much Fewer Resources</td>
<td>one_perfect_evals_lowest_resources\</td>
</tr>
</tbody>
</table>

### Table #2: Algorithm testing data file locations

The following table provides the location (on the CDR) of the data files for the printer simulations that were explained in section 6.2.

<table>
<thead>
<tr>
<th>Simulation</th>
<th>Location (base directory: \Experimentation\printer_sims)</th>
<th>Main Program Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simulation #1: Only One Printer Has All the Necessary Features</td>
<td>\test_one_good_printer\</td>
<td>test_one_good_printer.cc</td>
</tr>
<tr>
<td>Simulation #2: Same Job to Similar Printers</td>
<td>\test_same_job\</td>
<td>test_same_job.cc</td>
</tr>
<tr>
<td>Simulation #3: Same Job to Printers with Changing Evaluations</td>
<td>\test_change_rating\</td>
<td>test_rating.cc</td>
</tr>
<tr>
<td>Simulation #4: Different Printers with Changing Job Specifications</td>
<td>\test_change_job\</td>
<td>test_change_job.cc</td>
</tr>
</tbody>
</table>

### Table #3: Printer simulation data file locations
Bibliography


