BUILDING INTELLIGENT MARKET PLACES WITH SOFTWARE AGENTS

By

JAGADHA SIVAN

A THESIS PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

UNIVERSITY OF FLORIDA

2000
To my family
ACKNOWLEDGMENTS

I would like to express my sincere gratitude to my advisor, Dr. Joachim Hammer, for giving me an opportunity to work on this interesting and challenging project and also for providing continuous guidance, advice and support throughout the course of my work.

I thank Dr. Sherman Bai for his invaluable help and fruitful discussions that we had during the course of this work. I thank Dr. Herman Lam for serving on my supervisory committee and for his careful perusal of this thesis.

I would like to thank Ms. Sharon Grant for maintaining a great research environment at the Database Systems Research and Development Center.

On a more personal note, I would like to thank my family whose love, support and constant encouragement was of great importance through this work.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>ACKNOWLEDGMENTS</th>
<th>iii</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIST OF TABLES</td>
<td>vi</td>
</tr>
<tr>
<td>LIST OF FIGURES</td>
<td>vii</td>
</tr>
<tr>
<td>ABSTRACT</td>
<td>viii</td>
</tr>
<tr>
<td>CHAPTERS</td>
<td></td>
</tr>
<tr>
<td>1 INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.2 The Market Place</td>
<td>4</td>
</tr>
<tr>
<td>1.3 Thesis Outline</td>
<td>8</td>
</tr>
<tr>
<td>2 RELATED RESEARCH</td>
<td>10</td>
</tr>
<tr>
<td>2.1 Introduction</td>
<td>10</td>
</tr>
<tr>
<td>2.2 Software Agents</td>
<td>10</td>
</tr>
<tr>
<td>2.3 The Aglet Model</td>
<td>15</td>
</tr>
<tr>
<td>2.4 Database Retrieval</td>
<td>17</td>
</tr>
<tr>
<td>2.5 Data Format Conversion</td>
<td>20</td>
</tr>
<tr>
<td>2.6 Conclusion</td>
<td>25</td>
</tr>
<tr>
<td>3 THE E-MARKETPLACE</td>
<td>27</td>
</tr>
<tr>
<td>3.1 Introduction</td>
<td>27</td>
</tr>
<tr>
<td>3.2 The Client Component</td>
<td>29</td>
</tr>
<tr>
<td>3.3 The Server Component</td>
<td>33</td>
</tr>
<tr>
<td>4 DETAILS OF THE MARKET PLACE</td>
<td>37</td>
</tr>
<tr>
<td>4.1 Introduction</td>
<td>37</td>
</tr>
<tr>
<td>4.2 Software Agents: Creation, Mobility and Messaging</td>
<td>37</td>
</tr>
<tr>
<td>4.3 Client Side Implementation</td>
<td>45</td>
</tr>
<tr>
<td>4.4 Server Side Implementation</td>
<td>47</td>
</tr>
</tbody>
</table>
## LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5-1: Database entries for ISBN 16</td>
<td>65</td>
</tr>
<tr>
<td>5-2: Constraint sets and the respective results.</td>
<td>66</td>
</tr>
</tbody>
</table>
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1: The Users View of the e-market place</td>
<td>5</td>
</tr>
<tr>
<td>1.2: Architectural components of the market site</td>
<td>7</td>
</tr>
<tr>
<td>2.1: Mobile Agent Architecture</td>
<td>12</td>
</tr>
<tr>
<td>2.2: Relationships between Host, Server Process, Contexts, Poxies and Aglets</td>
<td>16</td>
</tr>
<tr>
<td>3.1: Components of the electronic market place involved in the first phase</td>
<td>28</td>
</tr>
<tr>
<td>3.2: Client site architecture</td>
<td>30</td>
</tr>
<tr>
<td>3.3: Server/Market Site Architecture</td>
<td>34</td>
</tr>
<tr>
<td>4.1: Sample price vs days distribution</td>
<td>59</td>
</tr>
</tbody>
</table>
Abstract of Thesis Presented to the Graduate School of the University of Florida in Partial Fulfillment of the Requirements for the Degree of Master of Science

BUILDING INTELLIGENT MARKET PLACES WITH SOFTWARE AGENTS

By

Jagadha Sivan

December 2000

Chairman: Joachim Hammer
Major Department: Computer and Information Science and Engineering

Electronic commerce has seen an explosive growth in the recent past with the business-to-consumer model being the most popular among Internet users. Here vendors and retailers provide a virtual shop for consumers to purchase items online. This has brought about a great increase in the consumer’s convenience to shop. However, the consumer still has to go to every vendor site to find the best bargain for an item. The next step in electronic commerce suggests that the consumer be allowed to go to only one site that gives him the best results from multiple vendors in order of his priorities.

The goal of this thesis is to design such a system. Our electronic marketplace allows multiple vendors to be registered on it. A user specifies his requirements, which is then satisfied by searching across the entire marketplace arriving at the best possible fit. If exact matches are not found, the next best options are returned to the user by dynamically improving the intelligence of search algorithms. The use of software agents to implement the marketplace adds robustness and scalability to the system.
No revolution has held more promise and has broken more barriers than the Internet revolution. From about a million Internet connections in the early 1990s to hundreds of million now, the Internet is no longer just a hugely connected repository of organized information but a powerful platform for the next generation commerce or Electronic Commerce (e-commerce). Currently the most common and widespread form of electronic commerce is the Business to Consumer or B2C model [B2C00], where the businesses directly reach out to the consumers by means of their online websites. Consumers navigate through the site filling their shopping cart much like in a real store. When finished, the consumer makes an electronic payment for everything in his\(^1\) cart and the products get shipped to him. Amazon.com [Ama99] and drugstore.com [Dru99] are some examples of popular B2C sites.

While the popularity of the B2C web-sites is increasing among the lay consumers, the real promise of the Internet lies in the even more promising Business to Business or B2B model [B2B00]. B2B is an automated way of how companies collaborate across a supply chain with their customers, suppliers and partners. The concept itself is not totally new, but has been around for a while in Electronic Data Interchange (EDI)

\(^1\) For brevity, references to she/he, his/hers, her/him were condensed to he, his and him respectively. Readers are requested to read the condensed versions exclusive of historical or social context.
However, it is the use of common communication protocols, standardized infrastructure and the ubiquitous nature of the Internet that has allowed the business-to-business automation to become such a lucrative proposal for businesses. To assert this point with some statistics, consider that in 1999 consumers bought $7.8 billion worth of products online. By 2003, consumers are expected to buy $108 billion worth online. On the other hand, last year the B2B market was five times the $7.8 billion purchased by consumers, and by 2003, B2B commerce will generate $1.3 trillion in revenues—12 times what it expects of the consumer market [Web00].

The huge increase in numbers of online customers and businesses generate a new set of issues such as scalability and robustness of systems. With only performance in mind, however, it becomes necessary to revisit traditional client-server architectures and arrive at newer and better models. The thin-client [Sin99] and the software agent [Klu99] model are among the two newer architectures that are helping meet the challenges of these new distributed systems. The thin-client essentially represents a three-tier architecture (as opposed to the two tiers in a client server model). The client layer (first layer) is a simple browser and the third tier is the database server. The core workload of the system is borne by the middle tier. This layer has the business logic and rules and all processing and refining of data take place here.

It may be noted that the thin client requires a lot of data moving back and forth between the client and the server on the network. This not only increases the network traffic, but also increases the dependency of the system performance on the latency of the network. Software agents directly address this issue by making use of the philosophy of “moving the code to the data, instead of the data to the code.” Software agents are
programs that can suspend execution on a system, transfer themselves to a remote system and continue execution there. Necessary state and environment information is carried along by the agent itself. The virtual platform on which these agents run allows them to see a consistent interface on any hardware and execution at the remote location resumes uninterrupted. Moving the processing to the data allows the system performance to be unaffected by the network loads and only the end results need traverse the network.

The Java language [Sun99] is the perpetrator of portable or the “write once run anywhere” code. As a result, Java has been partially responsible for fueling the phenomenal growth and popularity of the Internet. Needless to say it is the language of choice to implement both the thin-client as well as the software agent models. Enterprise Java Beans (EJB) [Mon00] is a popular thin client software platform and is part of the Java 2 Enterprise-Edition (J2EE) by Sun Microsystems. Among the more popular Java software agents, we have Voyager [Obj99a] by Object Space [Obj99b] and Aglets [Agl99] by IBM [IBM99].

Initially, the Internet was intended mainly for the purpose of information retrieval and its underlying HTTP protocol for transporting HTML documents was sufficient to display information on web browsers. The emergence of commerce on the Internet requires some more attention in standardization, mainly in document exchange. The flow of goods through the manufacturing process, for example, begs for automation. But schemes that rely on complex, direct program-to-program interaction have not worked well in practice as they depend on a uniformity of processing that does not exist. For centuries, humans have successfully done business by exchanging standardized documents: purchase orders, invoices, manifests, receipts and so on. Documents work for
commerce because they do not require the involved parties to know about the internal procedures of the other partners. Each record exposes exactly what its recipient needs to know. The exchange of documents is way to do business on-line too. But this was not the job for which HTML was built. The answer is XML, which in contrast was designed for electronic document exchange, and it is becoming clear that universal electronic commerce will rely heavily on a flow of agreements expressed in XML documents streaming through the Internet. XML-powered Web will be faster, friendlier and a better place to do business.

1.2 The Market Place

Typically, a market place is a place where people meet for the purpose of trade by private purchase and sale. With today’s electronic communications and computer technology it is possible for people to trade and conduct business online. This is what e-markets are about. The idea behind e-markets is to replace physical business transactions with electronic business transactions using the Internet. Most businesses support e-commerce because processing is faster and relatively error-free. Additionally, as all purchase orders are made electronically, costs are drastically reduced. On the consumer front, it is convenient and easy to use. Consumers can shop for any product, from pet food to airline tickets to cars, from the convenience of their homes and offices.

In our thesis, we have created a new improved e-market place. Our e-market place is a conglomerate of different market sites. Market sites can exist independently (legacy sites [Leg00]) or may have been created specifically to participate in our e-market place. Later on, we discuss the requirements that must be met by each site to participate in our e-market place. Each of these market sites has a number of suppliers registered with it.
Typically, the consumer himself would like to exhaustively search through all the market sites to get the best deal. However this is time consuming. We address this issue by building an agent-based infrastructure where a user can simply state his request and have agents automatically search the e-market place for the best deals.

Figure 1.1 depicts the user’s perspective of our market place. As discussed, the market place has a number of market sites, each having a set of suppliers registered with it. Each market sites needs to be registered with the market place. This is done at the market place registry. We will discuss the global DTD present at the registry in a later section.

Figure 1.1: The Users View of the e-market place
Agents are sent out to search the market sites. Each of these agents return the results to the client site when are then merged and displayed to the consumer.

Figure 1.2 shows the architectural components of the market place. At each market site, a user’s request is processed in two phases. In the first phase the market sites are searched to find suppliers who can fulfill the users constraints. In case the constraints are not exactly fulfilled, they are relaxed providing an intelligent set of next closest options. As an example, suppose the user requests 100 copies of a particular book for a certain price and requires them in 3 days. In the absence of results matching the constraints exactly, the system will find suppliers who closely match the constraint set i.e., find suppliers who can supply lesser or more than 100 books or whose quote is more than the requested price. This relaxation, which is described in more detail in later sections of this thesis, requires user input for best results. At the end of this phase, the consumer is provided a list of suppliers who mat his choice the closest. The user can further set up negotiations with any or all of these suppliers. These negotiations constitute the second phase.

To negotiate, first the customer formulates a negotiation strategy based on factors in his requirements that are flexible and those that are not. The consumer negotiation agent is informed about this. It then creates a market floor consumer (MFC) negotiator for every supplier that the consumer wants to negotiate with.

The consumer negotiation agent then informs the supplier negotiation agent about the suppliers that the consumer wants to negotiate with. The supplier negotiation agent then creates market floor supplier (MFS) negotiators for every supplier and packs them with the negotiation strategy of the supplier.
Figure 1.2: Architectural components of the market site

The market floor consumer and supplier negotiators negotiate at the neutral negotiation server that each market site has access to. Each market floor consumer negotiator is in contact with other siblings, exchanging notes on how each is faring. For instance if one MFC negotiator was getting a faster delivery date for a certain price, this is spread across other MFC negotiators so they can all dynamically update their negotiation strategies. In a scenario where there are many market sites each hosting a different group of Suppliers, MFC negotiators in each market site can communicate across market sites. When the negotiators finalize a deal, the consumer and the chosen
supplier are informed. The consumer and the supplier then send their confirmation or non-confirmation to the verifying agent, which directs them across to the respective supplier and consumer. If the verifying agent receives positive confirmation from both, the supplier and the consumer, the deal is considered to be ‘sealed.’ The database controller is contacted to update the database. Also, the other suppliers who were negotiated with are given details of the final offer made to the consumer. While doing this, the details of the chosen supplier are not disclosed, nor are the details of the other suppliers who were part of the negotiation process.

The updating agent addresses all database updating issues that the suppliers would encounter, whether it be change in address, stock updating or a change in the negotiation strategy that is stored in the database. But, the actual updates to the database are still carried out by the database controller.

Hence at the market place, the consumer can state his requirements and negotiate with suppliers who can match his requirements. This enables consumers and suppliers to arrive at a natural market driven value. This thesis offers a detailed description of the design and implementation issues of the first phase. Although some design details of the negotiation stage will be mentioned, where necessary, the specifics are beyond the scope of this thesis.

1.3 Thesis Outline

As stated earlier, this thesis concentrates on the first or the pre-selection phase of the market place operation. Chapter 2 provides related background information outlining the various technologies for implementing electronic market places including mobile software agents. Chapter 3 describes the details of our approach, focussing on the
architecture of our market place and the design decisions we had to face along the way. Chapter 4 highlights the implementation which is based on mobile software agents. Some ideas for query relaxation are also provided in this chapter. In chapter 5 we provide our test bed and analyze the results. Finally, Chapter 6 summarizes our work along with suggestions on future expansions.
CHAPTER 2
RELATED RESEARCH

2.1 Introduction

The previous chapter introduced the underlying concepts of the electronic market place. This chapter reviews the existing technologies that are available to us to build a workable model of our market place.

Instead of following the conventional client-server model to implement our electronic market place prototype system, we are experimenting with software agents [Gen94] for transporting data back and forth between the market place and its users (i.e., the consumers). The properties and advantages of software agents are explained in the first section. Among the different software agents that are available, our preferred model is the Java-based Aglet model [Agl99]. We examine at the various features and advantages in using this model. Later sections explain the idea and benefits of query relaxation procedures. The techniques itself are dealt with in a later chapter. Finally, we end the chapter by covering the topic of XML and its importance in today's e-business world. Specifically, we describe important concepts and technologies and how XML and its related components form a powerful data representation package.

2.2 Software Agents

Software agents are a relatively new approach for designing, implementing and maintaining a distributed system. The agent model allows for the creation of highly
robust and fault-tolerant systems. As opposed to the client server model where the server has a set of known services and the client has to work with the services provided by the server, the agent model allows any host to scale its level of information retrieval by building a suitable agent that can transport itself to the server to perform the necessary work.

There are two types of software agents viz. stationary agents and mobile agents. As the names imply a stationary agent executes only on a single system during it's life cycle, while a mobile agent can move from one host to another, in a network, carrying state and environment information with it. Agents operate asynchronously and are independent of the process that created them. Clever designs of mobile agents reduce network traffic and provide an effective means of overcoming network latency.

The life cycle of a software agent comprises a life-cycle model, computational model, a security model, and a communication model [Gre97]. A mobile agent additionally defines a navigation model. Services to create, destroy, suspend, resume and stop agents are needed to support the life-cycle model. The computational model refers to the computational capabilities of an agent, like data manipulation. The security model describes the ways in which agents can access network resources, as well as the ways of accessing the internals of the agents from the network. The communication model defines the communication between agents and between an agent and other entities, like the network. All issues referring to transporting an agent from one location to another are handled by the navigation model.

The basic architecture of the mobile agent system is shown in the figure below. The agent system resides on the operating system and is platform neutral i.e. it offers a
virtual platform to all its agents, independent of the underlying hardware or operating system. This is the basis for location-independent agent execution. A stationary or mobile agent residing on the agent system has associated with it an identifier, a state, an environment and an interface for communication. Agents communicate using the communication infrastructure provided. This infrastructure sits on top of the underlying transport layer providing the agents with a virtual layer on top of the actual network.

Figure 2.1: Mobile Agent Architecture

Advantages of Mobile Agents

Mobile agents as indicated earlier are software programs that can suspend execution, transport itself to another host along with its environment and state information and resume execution on the new host. Apart from those mentioned above, the following are some more advantages of mobile agents [Lan98].

Reduced network load and latency: Distributed systems involve communication protocols and multiple interactions between two (or more) hosts to accomplish a given
task. Mobile agents allow a conversation to be packaged and sent to the destination host where the interactions can take place locally. Thus, unlike traditional models which move data to the computation, mobile agents move the computation to the data. This not only reduces the network traffic but also the latency.

**Support for heterogeneous environments:** Although the network computing environment is heterogeneous, as such mobile agents are generally computer and transport layer independent and dependent only on their execution environment. This provides optimal conditions for seamless system integration.

**Asynchronous and autonomous execution:** Mobile agents do not require a continuous connection between machines or processes. Once dispatched, the agents are independent of the creating process and can operate asynchronously and autonomously.

**Robustness and fault tolerance:** A mobile agent can react dynamically to unfavorable situations. If a distributed system begins to malfunction, all agents executing on that machine will be given time to dispatch and continue their operation on another machine.

**Applications of Mobile Agents**

Mobile agents are well suited for data-intensive applications, where the data is remotely located, owned by a remote service provider and the user has specialized needs. Another area is extensible servers, where the user can ship and install an agent representing him more permanently on a remote server. This makes the agent like a personalized, autonomous piece of code that runs remotely and contacts the user when events of interest occur. Given that mobile agents can create clones in the network, another potential use of mobile agent technology is to administer parallel processing
tasks. Some other uses include network wireless network packet routing [Min99], network management [Bie98], client-server networking alternatives [Mul98], and electronic commerce.

**Mobile Agent Systems**

Now we discuss some of the more popular mobile agents. Java has generated a flood of mobile agent systems. The property of Java offering a virtual platform independent machine to create applications, lends itself naturally for creating agent systems. However languages like Tcl [Sch98] and Python [Bea98] are also in use. The following are some of the mobile agents.

**Concordia, Mitsubishi Electric Information Technology Center America:**
Concordia [Mit99] is a full-featured framework for development and management of network-efficient mobile agent applications for accessing information anytime, anywhere and on any device supporting Java. With Concordia, applications can process data at the data source; process data even if the user is disconnected from the network; access and deliver information across multiple networks (LANs, Intranets and Internet); use wire-line or wireless communication; support multiple client devices such as Desktop Computers, PDAs, Notebook Computers, and Smart Phones.

**Voyager, Object Space:** Voyager is a 100% Java agent-enhanced Object Request Broker (ORB). It combines the power of mobile autonomous agents and remote method invocation with complete CORBA support and comes complete with distributed services such as directory, persistence, and publish subscribe multicast. Voyager allows Java programmers to quickly and easily create sophisticated network applications using both traditional and agent-enhanced distributed programming techniques.
Agent TCL, Dartmouth College: Agent TCL [Dag99] is a mobile system whose agents can be written in TCL. In addition to migration, Agent TCL supports Message passing - agents can pass messages to each other; Rudimentary security - Any agent, message or connection request that does not come from an approved machine is ignored. The system administrator specifies the set of approved machines; Generic timeout and retry mechanisms - Agents can retry a command as many times as desired and can impose a timeout on arbitrary TCL code.

Aglets, IBM Japan: An Aglet is a Java object that can move from one host on the Internet to another. That is, an aglet that executes on one host can suddenly halt execution, dispatch to a remote host, and resume execution there. When the aglet moves, it takes along its program code as well as its state. A built-in security mechanism makes it safe for a computer to host untrusted aglets. Aglets were our agent of choice in setting up our e-Marketplace. Our reason for choosing Aglets, their architecture and working will be discussed in a later chapter.

2.3 The Aglet Model

The software agents in our system were implemented using aglets. The following features needed for our electronic market place were satisfied by the aglet model.

Mobility: Aglets provide a very simple Java API to implement the mobility feature.

Autonomy: Aglets can be programmed to make intelligent decisions, execute code and also move to and run on any machine that supports them. Additionally, the route traversed by an aglet can be preset, and also be dynamically modified as it discovers additional information during its journey.
Response time: Aglets have rapid response time [Das99]. They can visit several sites, negotiate with local software at each site, and can return to their home base in order of a few seconds.

Concurrency: Multiple aglets with similar objectives can be dispatched simultaneously to accomplish various parts of a task in parallel.

Local Interaction: Mobile aglets can interact with local entities, such as databases, file servers and stationary aglets, through method invocation, and with remote entities is by message passing.

![Diagram of relationships between Host, Server Process, Contexts, Poxies and Aglets](image)

Figure 2.2: Relationships between Host, Server Process, Contexts, Poxies and Aglets

Aglets can run on any machine that supports the Aglet API. They are hosted by an aglet server, similar to the way applets are hosted by a Web browser. The aglet server provides an environment for aglets to receive and host aglets. The key abstractions in the Aglet API are the context and the proxy.
Every aglet also has a unique identifier and resides in a context, which is its workplace. The server could host multiple contexts, each of which may host one or more aglets. The context provides a means for maintaining and managing the running aglets in a uniform execution environment. The proxy is a representative of an aglet. To interact with an aglet one must do it via the proxy. The proxy also provides location transparency i.e., an aglet and its proxy need not be at the same location, allowing the local proxy to hide the remoteness of the aglet.

2.4 Database Retrieval

Every marketplace has a database, which is a repository of information about the products, services, suppliers, etc. When a customer states his requirements, the database is accessed and searched for matches. In an ideal situation, the database would have an exact match or the required number of exact matches, depending on the number of results the customer has requested. But in many cases, it may not be possible to find exact matches. Better than leaving it to the customer to change the constraint set and query the database again, we could provide a better approach by relaxing some constraints and showing him his next closest choices.

Structured Query Language (SQL) [Sil98] is a language used to create, manipulate, examine and manage relational databases. SQL was standardized across different database vendors so that a program could communicate with most database systems without having to change commands. Open database connectivity (ODBC) [Mic98] provides a consistent programming language SQL interface for communicating with a database. Using ODBC and SQL, we can connect to a database and manipulate it in a standard way. The popularity of Java as the “write once run anywhere” language has
soared in the last few years. Java provides a library for database connectivity called the Java Database Connectivity (JDBC) [Ree97].

JDBC is the Java application programming interface (API) for standardized SQL based database access. It is a database independent API that facilitates development of databases independent “write once run anywhere” Java applications.

Current query processing requires that queries be specified precisely and thus requires users to fully understand the problem domain, the database structure and content, while providing limited answers and options, or even no information at all, if the exact answer is not available. The solution is to extend the classical notation of query answering to cooperative query answering [Cup89].

Cooperative query answering provides neighborhood or generalized information relevant to the original query and within the semantic distance of the exact answer. A cooperative query answering process consists of enlarging the scope of a query by relaxing the searching range and refocusing to the nearest subranges of the original query. To carry out this relaxation, various levels of abstraction and refinement could be pre-defined or the relaxation could be performed dynamically.

Currently, CoBase [Chu96] is one of the few systems that support cooperative query answering. CoBase, is a cooperative database system that can derive approximate answers to a query by relaxing query conditions when no exact answers are available. It utilizes a knowledge structure--type abstraction hierarchies (TAHs) [Chu94] - that provides multi-level knowledge representation of domain knowledge. On top of the existing data schema, a type hierarchy specification is provided, based on the abstraction notion. The corresponding supertype and subtype domain values are stored in a table.
Based on the user’s interests, the set of queries and the expected answers, the desired TAH is constructed in the selected problem domain. Operations are provided to traverse the hierarchies, such as generalization or abstraction and specialization or refinement. Query conditions are relaxed to their semantic neighbors until CoBase can produce approximate answers. In systems like CoBase, the levels of abstractions and refinement are pre-defined.

In a scenario like a marketplace, where there are thousands of products, and each falls under a different price range, it is impractical to apply the above approach of predefining the various levels of abstraction and refinement. Also, the user of the marketplace may state the quantity he desires and the number of days he wants it in, the number of options he wants, etc. It is very expensive to pre-define the levels of abstraction and refinement for each product offered at the marketplace, and also for each of the stated constraints. It is more reasonable to perform a generic relaxation of the query. And this relaxation process continues till the required result is obtained or till all possible options at the database are exhausted.

Depending on the user’s preferred values, the boundary values he has stated, both in his constraint set, and values in the database, the constraints are relaxed. They are relaxed such that the boundary values stated by the user are never exceeded. The three main constraints in our market place are price, quantity and number of days to deliver. Price and quantity are relaxation by a percentage of the difference between the required value and the upper boundary value. At times, with these relaxations, there are instances when the number of results returned is more than required. In this case, the refinement
process selects the options that match closest to the user's stated requirements are selected. The method we have implemented is described in further detail in Chapter 4.

2.5 Data Format Conversion

In the real world, different computer systems and databases contain data in different and in most cases, incompatible formats. One of the most time consuming challenges for developers has been to exchange data between such systems over the Internet. XML (EXtensible Markup Language) is a standard being developed to overcome such issues. In this section we find out more about XML and other software that enable data exchange between incompatible systems.

XML

XML describes structure of data and focuses on its semantics. XML was created so that, richly structured documents could be exchanged. The only viable alternatives, HTML and SGML, are not practical for this purpose. HTML does not support arbitrary structure. SGML provides arbitrary structure, but is too difficult to implement for a web browser.

XML is a set of rules for defining semantic tags that identify the different logical parts of a document. It is also a markup language that defines syntax for defining other domain-specific, semantic, structured markup languages. XML markup describes a document’s structure and meaning. It does not describe the formatting of elements, which can be done using a style sheet. Hence, XML documents contains tags to say what is in the document, not what the document looks like. Tags are created as and when required.
These tags must then be organized according to certain general principles. The tags can be documented in a Document Type Definition (DTD) [Har99]. We will discuss DTD’s shortly. A XML document is “well formed” if it conforms to the XML syntax rules. An XML document is “valid” when it is “well formed” and also conforms to the rules of the corresponding DTD.

The unifying power of XML arises from a few well-chosen rules. One is that tags almost always come in pairs. Like quotation marks, tag pairs can be nested inside one another to multiple levels. The nesting rule automatically forces a certain simplicity on every XML document, which takes on the tree structure. Another source of XML’s unifying strength is its reliance on a new standard called the Unicode [Uni00], a character-encoding system that supports intermingling of text in all the world's major languages. In HTML, as in most word processors, a document is generally in one particular language, whether that be English or Japanese or Arabic. If the software cannot read the characters of that language, then the document cannot be used. But software that reads XML properly can deal with any combination of any of these character sets. Thus, XML enables exchange of information not only between different computer systems but also across language boundaries.

Some of the advantages of using XML include the following:

- XML is ideal for large and complex documents because the data are structured. XML not only lets the programmer specify a vocabulary that defines the elements in the document, it also lets him specify the relations between elements.

- XML allows the design of domain-specific markup languages i.e., XML allows various professions to develop their own specific mark-up languages. This allows
individuals in the field to trade data and information without worrying about whether the
person on the receiving end has the particular proprietary payware that was used to create
the data.

- XML is non-proprietary and easy to read and write. As a result, it is an
good format for the interchange of data among different applications. One such
format under current development is open Financial Exchange Format (OFX). OFX is
designed to let personal finance programs, like Microsoft Money and Quicken, trade data.

**DTD**

XML provides an application independent way of sharing data. Independent
groups of people can agree to use a common DTD for interchanging data. The application
can use a standard DTD to verify that data that is received from the outside is valid. It can
also use a DTD to verify its own data. The purpose of a DTD is to define the legal
building blocks of an XML document. It specifies a set of rules for the structure of a
document. DTDs also help ensure that different programs can read each other’s files. The
DTD defines exactly what is and what is not allowed to appear inside a document. A
DTD shows how the different elements of a page are arranged without actually providing
their data. DTDs can be included in the file that contains the document they describe, or
they can be linked from an external URL. If a DTD is applied to multiple documents, a
URL can be used to specify precisely where the DTD is found.

Hence, DTDs provide a means for applications, organizations, and interest groups
to agree upon, document and enforce adherence to markup standards. A large number of
forums are emerging to define standard DTDs for almost everything in the areas of data exchange [Com99].

DOM

The Document Object Model (DOM) [DOM99] is a platform- and language-neutral Application Programming Interface (API) that allows programs (e.g., XML, HTML) to dynamically access and update the content, structure and style of documents. The document can be further processed and the results of that processing can be incorporated back into the original document.

Increasingly, XML is being used as a way of representing many different kinds of information that may be stored in diverse systems, and much of this would traditionally be seen as data rather than as documents. Nevertheless, XML presents this data as documents, and the DOM may be used to manage this data. With the DOM, programmers can build documents, navigate their structure, and add, modify, or delete elements and content. The Document Object Model specifies a tree-based representation for an XML document. The DOM is useful for modifying XML documents, as functions are defined that allow a programmer to create a DOM tree, traverse it, access element and attribute values, modify the tree by adding new nodes, moving subtrees around or deleting nodes, and also produce a new XML document as output. As a W3C specification, one important objective for the DOM is to provide a standard programming interface that can be used in a wide variety of environments and applications. DOM is designed to be used with any programming language.
The eXtensible Stylesheet Language (XSL) [XSL00] is a formatting and transformation language i.e., it comprises two separate XML applications, for transforming and formatting XML documents respectively. As the name suggests, the transformation language provides elements that define how one XML document is transformed to another and the formatting language is used to describe how the content should be rendered when presented to the user. Since our e-market place relies on the ability to convert XML documents from one format to another using XSL, we discuss this in further detail.

The XSL transformation language operates by transforming one XML tree into another XML tree. The language contains operators for selecting particular nodes from the tree, reordering the nodes, and outputting nodes. Both the input and the output must be XML, HTML or SGML documents.

An XSL document contains a list of template rules. A template rule has a pattern specifying the trees it applies to and a template to be output when the pattern is matched. When an XSL processor formats an XML document using an XSL stylesheet, it scans the XML document looking through each subtree in turn. As each tree in the XML document is read, the processor compares it with the pattern of each template rule in the stylesheet. When the processor finds a tree that matches a template rule's pattern, it outputs the rule's template. This template could include some new markup, some new data and some data copied out of the tree from the original XML document.

There are three primary ways by which XML documents can be transformed into other formats with an XSL stylesheet:
1. The XML document and associated stylesheet are both sent to the client site, which then transforms the document specified by the stylesheet and presents it to the user.

2. The server applies an XSL stylesheet to an XML document to transform it to some other format and sends the new document to the client.

3. A third program transforms the original XML document into some other format. This third program may be at the client site, server site or at any other remote site.

Each of these approaches uses different software. We are interested in the second method. In the e-marketplace, the server needs to apply an XSL stylesheet to an XML document to transform it to another XML document. There are several processors available that accomplish this task (e.g., Xalan [Xal00a], Saxon [Sax00], Koala XSL Engine [Koa00]). Xalan was used. Xalan is feature-rich and robust. Unlike most other processors that can be used from the command line only, Xalan can also be used in an applet or a servlet, or as a module in other program. The input may be in the form of a file, character stream, byte stream, DOM or SAX input stream. Xalan performs the transformations specified in the XSL stylesheet and produces a document file, a character stream, a byte stream, a DOM or a series of SAX events, as specified when the transformation was setup.

2.6 Conclusion

Software agents believe in moving code to the remote data, rather than data to the code. This potentially has huge bandwidth savings and can overcome network latency. Among the many available software agent packages, the Java-based aglet package has a few attractive features. Aglets provide a very powerful, yet simple API that allows for
quick implementation and easy deployment. Query relaxation provides us the means of helping the user get close matching results in the absence of exact matches. This helps the user know what his next best options are, instead of him explicitly making such request. XML enables the data exchange between various incompatible systems over the Internet and allows for structured content delivery over the web.
CHAPTER 3
THE E-MARKETPLACE

3.1 Introduction

An electronic market place gives the consumer the ability to access and compare the products and services of multiple sellers online through a web browser. When a consumer enters the e-market place, the operations that follow can be roughly be classified into two phases. The sequence of events from when the consumer enters his request to when the preliminary results of the supplier search are displayed can be considered the first phase. Specifically, this includes his request getting sent to the various relevant market sites via mobile agents. The results that the agents bring back contain names of suppliers who can match the requested quotes.

The second phase is the negotiation phase where the consumer selects some or all of the suppliers returned to him. He then creates a negotiating strategy and negotiates with any or all the suppliers from the list returned to him, to arrive at an optimal result.

As stated earlier, this thesis offers a detailed description of the design and implementation issues of the first step. Although some of the design details of the negotiation stage will be mentioned, where necessary, the specifics are beyond the scope of this thesis and are the subject of research of another team member.

The e-market place consists of a number of different e-market sites. The architecture of the electronic market place can be broken up into two main components viz. the client component and the server or market site component. Additionally, the
market place also contains a global DTD. The chief purpose of this DTD is to help define rules to carry out translations between the XML requests and the information available at the different market sites. Consumer requests are in the form of XML documents, which need to be translated into requests that can be understood at each market site. As is typical in many systems, the schema followed by the different components comprising the e-market place may not be the same. Trading data at each site is stored in a database. Though the databases at each of the market sites contain similar information, it is impractical to assume that every database follows the same schema. Figure 3.1 depicts the various components involved in the pre-selection phase.

Figure 3.1: Components of the electronic market place involved in the first phase
XSL (eXtensible Stylesheet Language) serves to bring about a common ground to correlate these different schemas, as expressed by the global DTD (defining user requests) and local DTDs (defining the database at each market site). Every market site defines XSL rules to convert from the consumer site format to the format followed by the local database and vice versa. An XML format converter at each site stores and retrieves the correct XSL rules when an incoming agent is submitting a new request.

The global DTD can be at any location, as long as it can be accessed by the XML format converter on the server side. Typically, it is referenced by specifying the URL of the site on which it resides. Now we go on to explain the main parts of the client and server components.

### 3.2 The Client Component

The client component provides the user with a window into the electronic market place. The main role of the client component is to accept user requests, package them in the XML format specified in the global DTD, dispatch mobile shopping agents, carrying the request, to the various market sites and display the results that have been returned. The main elements comprising the client component are shown in Figure 4.2. We now discuss the functions of each of these elements.

**User Input Interface**

This is a GUI (Graphical User Interface) that allows the user to specify his requirements. It is an application running on a web browser. The requirements collected from the user are passed (in text format) to the request processor.
The request processor is the main coordination and processing component of the client component. The user requirements from the input interface are converted to an XML format. This newly created XML document is checked to see if it conforms to the structure of the global DTD. The next step is to dispatch the user requests packaged in XML to each of the market sites that comprise the market place. The request processor has a list and address of all such market sites and it communicates with each of these sites using mobile agents called shopping agents. The request-XML document is attached to each of these shopping agents. There is one shopping agent sent to each site that may offer relevant goods and services corresponding to the request. Once the requests are...
sent, the request processor listens on a pre-determined port for the results from the
shopping agents. The results are in an XML format. The request processor then merges
these XML documents, creating a results-XML document. The result-XML document
converted to HTML format and displayed to the consumer.

**Shopping Agent**

Shopping Agents are mobile agents created and spawned by the request processor
to fetch information from the different market sites. Each spawned shopping agent goes
to one site only. The address of the site it has to visit and the consumer’s requirement-set
are packed into the shopping agent. The shopping agent travels across the network and
upon reaching the server site, announces its arrival to the receiving agent at the site. It
then passes on the consumer’s constraints to the server and waits until it receives the
result from the server site. This result is also in an XML format. After receiving the
result, the shopping agent returns back to the client site and passes the result on to the
request processor. After transferring the result to the request processor, the Agent dies.

Thus, per user request, if there are ‘n’ server sites to be queried for information,
then ‘n’ shopping agents will be created, one per market site. It was initially decided to
have only one shopping agent per user request. Along with the consumer’s request set,
this shopping agent would be given a complete list of server or market site URLs to be
probed. This set of URLs define the agent’s itinerary. Additionally, the itinerary would
also specify the sequence in which the server sites would be visited. Once dispatched, the
shopping agent would visit the first server site specified in the itinerary. Upon receiving
the result from this site it would dispatch itself to the next site in the itinerary. Any
information received from the second site would be appended to the existing result set and so on. When the shopping agent has visited the last server destination, it returns back to the client with its consolidated result set. At first, this approach sounded appealing since it means the client site has to spawn off and initialize only one shopping agent per user request. Even the elimination of duplicate information can be done on the fly as the agent hops from one server site to another, saving some processing for the client before it display these results.

We disregarded this approach for the following reasons. The shopping agent queries the server list sequentially. So, the results can be displayed only when the shopping agent returns after traversing all the server sites. In the current approach we can display interim results even before all of the dispatched shopping agents return.

As the shopping agent moves from one server site to another, the information it carries with it keeps growing to a potentially very large size. This not only increases the transfer time during subsequent hops, but also makes the overall transfer of information inefficient as a lot of information is carried from one site to the next. Additionally this approach would have to address potentially harmful scenarios of what must be done if any of the server sites in the itinerary were to stop responding, especially while the shopping agent was already resident on the server. This could lead to a loss in information already collected from the previous sites. However, in the approach where we dispatch multiple shopping agents a simple time-out mechanism could solve the problem.

From the above discussion it is clear that when multiple shopping agents are dispatched, the response time is simply the maximum of the response times of all the
server sites. However, in the case when only one shopping agent is dispatched the response time is the sum of the response times of all server sites. In the latter case, the extra effort to repetitively transfer previous results leads to an additional increase in the response time. It is for these reasons of performance, efficiency and scalability that we disregarded the approach of a single shopping agent.

Finally, before we end this discussion, it may be helpful to add that if the number of server sites increase to a very large number, there may be a crossover point where dispatching individual shopping agents may prove expensive. For true scalability, we may have to arrive at a fair compromise of the above two approaches and dispatch a shopping agent for every two or more server sites, as appropriate. Statistical comparison of these two approaches and identifying an approximate crossover point can be a subject of future work.

3.3 The Server Component

A number of server sites or market sites constitute the e-market place. The main role of the server site is to satisfy an incoming requirement request from a user. It queries its repository across multiple vendors while judiciously relaxing the requirements, if necessary. Each server site has a number of suppliers registered with it. A supplier could be registered with one or more market sites. In our current design, each server site runs an aglet server. The server performs several functions including creating aglets, receiving and dispatching them, and also retracting them, if necessary. The server site consists of the following components:

1. Receiving Agent: to receive shopping agents arriving from the client site
2. Searching Agent: to process the request from shopping agent just received
3. **XML Format Converter**: to convert between different XML formats

4. **Database Processor**: to convert incoming requests into SQL queries against the market site database and apply relaxation if necessary.

5. **Database**: contains information about the suppliers, their products, transactions, etc.

Figure 3.3 above shows the various components at each market site of the market place. We now describe the purpose and functions of each of these components.

**Receiving Agent**

The receiving agent is a stationary agent that is bound to a port at the server. Its main role is to receive incoming shopping agents and initiate processing. The receiving agent has a well-known port on which it listens for messages. This port number is listed at the market place registry that we discussed earlier. The receiving agent detects the arrival of the shopping agent and goes on to extract the request-XML document. It then spawns off a searching agent and passes on the consumer request to it. From this point...
on, communication between server and shopping agent is done via the newly created searching agent. The receiving agent no longer has a reference to the shopping agent and it continues to listen for other incoming shopping agents.

Searching Agent

The searching agent is a stationary agent that is created exclusively to process an incoming shopping agent’s request. When the searching agent is created, the ID or proxy of the corresponding shopping agent is passed on to it, along with the user request document in XML. The first step of processing is to convert this XML document to the format that conforms to the schema of the local database. The XML format converter is invoked to perform this task. Upon completion of this conversion, the new XML document is parsed by the searching agent resulting in a DOM (Document Object Model) tree. The DOM tree is then traversed to obtain the user constraints, which are communicated to the database processor to generate the SQL query.

The results matching the consumer’s requests are returned from the database processor and are converted to the XML format followed at the consumer end. Again, this conversion is carried out by the XML format converter. The resulting XML document is transferred to the waiting shopping agent and the searching agent then dies. The shopping agent is programmed to wait till it receives results from the market site.

XML Format Converter

As the name suggests, the XML format converter converts the request/results from one XML format to another. Every market site has a XML format converter that converts an XML document between formats followed by the consumer and the local database schema and vice versa using XSL. The searching agent invokes the XML format converter when it requires an XML conversion, which in turn invokes the XSL processor.
to carry out the conversion. Two sets of XSL rules are defined at each market site. The first document contains the consumer-to-database schema conversion specification and the second, the database-to-consumer schema conversion.

The XML format converter receives the XML document to be converted and a message from the searching agent stating whether the document is a consumer document or a server one. Depending on this message, the Converter uses the required XSL document and performs the conversion.

**Database Processor**

The database processor receives the set of consumer requirements from the searching agent. It forms an appropriate SQL query from these constraints, connects to the database and then queries it. When the results of the query are returned, they are checked to see if they are in accordance with the consumer’s requirements. Two problems may arise: the number of results is more than the consumer’s requirements or number of results is less than the consumer’s requirements. In the former case, the query is refined and the database is queried. In the latter case, the query is relaxed. This process of relaxation and refinement is repeated as required.

To prevent malicious and untrusted agents from viewing, tampering or corrupting data in the database, none of the agents in the market site are given permission to directly connect to the database. All data accesses and queries can be done only through the database processor. The database processor receives query requests from various agents. It first authenticates the agents and then executes the queries.
CHAPTER 4
DETAILS OF THE MARKET PLACE

4.1 Introduction

Chapter 3 provided an introduction to and an architectural overview of the various components that make up our electronic market place. Our system makes extensive use of software agents, both stationary and mobile. This chapter explains the details behind the approach in implementation of our electronic market place prototype, focusing on each component as well as the communication infrastructure.

We start the chapter by describing how we use aglets in accomplishing our objectives of building stationary and mobile agents. The next two sections describe the details of the client and server implementation with agents. XML conversions which are instrumental in bringing together diverse market sites in our system are explained in the next section, followed by our approach to relax user constraints to satisfy queries against an information repository.

4.2 Software Agents: Creation, Mobility and Messaging

The architecture of our electronic market place relies on our ability to automatically create and deploy software agents. Chapter 2 highlighted the numerous advantages to using software agents in a distributed system such as our e-market place and it also emphasized why aglets are our agent of choice. We will now discuss how we use aglets to implement our market place.
The Aglet Framework

The aglets framework consists of three layers viz. application, runtime and communication layer. The application layer is uppermost layer where the aglets defined by the users reside. The support provided to running aglets is in the runtime layer. This layer has three components viz. the persistence manager, for storing and retrieving deactivated aglets, the cache manager, for managing the resources used by the aglets and the security manager for protecting the hosts and the aglets. The runtime layer itself has no built-in mechanism for transporting aglets over the network but interfaces with the generic communication layer, which is the third layer.

The communication layer offers agent transport and communication mechanism independent of the underlying transportation mechanism. The current implementation of the communication layer uses Agent Transfer Protocol (ATP). ATP servers attempt to make direct connection to hosts in the network. Some networks are protected by a firewall that prevents users from directly opening socket connections to external nodes, because of which an aglet cannot directly be dispatched or retracted through a firewall. To overcome this, ATP supports HTTP tunneling which enables an ATP request to be sent outside the firewall as an HTTP POST request and the response is retrieved as an HTTP response.

When mobile aglets get dispatched to a new server, the Java object serialization mechanism is used to marshal and unmarshal the state of aglets into a stream. However on arrival at the remote host the aglet still needs the byte code to continue execution. One option is to transfer byte code from the source along with the aglet, or the other option is to let the aglet check the code available at the destination and retrieve the byte code from the source.
The aglet system has an environment variable, AGLET_EXPORT_PATH, which specifies a directory whose subdirectories are accessible from a remote host. All class files and other files located in these directories can be fetched from remote aglet servers.

**The Aglet API**

The aglet API is a Java package that is simple and flexible. It contains methods for creating and operating aglets, message handling, as well as dispatching, activating/deactivating, cloning and disposing of aglets. The *Aglet* class is the key class that provides the basis for creating customized aglets. Another important class is the *Message* class. Aglets communicate by exchanging the objects of the *Message* class. This class is discussed in the next section and later in this chapter. Other notable classes and interfaces include *AgletProxy*, *AgletContext*, and *AgletID*.

**Aglet Communication**

The principal way for aglets to communicate is by message passing. An aglet could potentially communicate with agents developed by other organizations. To support this, the aglets support an object-based messaging framework that is location independent and extensible. Several means of inter-aglet communication are supported including simple messaging with and without reply, advanced message management and multicast messaging between aglets.

Each message has two parts, type and data. The type helps to distinguish between messages and is set by the sender. Aglets may predefine their event scheme to listen in only to certain types of messages.
Aglet Security

The SecurityManager in the aglets runtime layer is responsible for protecting hosts and aglets from malicious entities. Every security sensitive operation requires consultation with the security manager. The SecurityManager component is based on the Java language system’s SecurityManager class.

An aglet has public methods that may be unsafe to expose directly to other aglets. So an aglet defines a proxy, which is a go-between to reach the aglet. The proxy is defined to allow only certain privileges to certain entities. More than one proxy can be defined to provide a different window of privilege to different types of aglets. Any aglet that wants to communicate with other aglets must obtain the proxy and then interact through this interface. Hence, the aglet proxy acts as a shield that protects an aglet from malicious aglets. When invoked, the AgletProxy object consults the security manager to determine whether the current execution context is permitted to perform the method.

Because of their autonomous behavior, aglets can define their own security policy and request all servers to honor it. For example, an aglet may define a policy that allows only aglets created by the same user to access it. Secondly, contexts and servers are responsible for keeping the operating system safe. The server protects the local resources, while the context is responsible for hosting visiting aglets. Each context on a server may define a different security policy. For example, a context that serves the database may allow aglets to access the database, while other contexts may not.

We will now talk about the different aglets and how we created them:

Stationary Aglets: The Aglet package of the aglet API makes available an Aglet class, which is an abstract class that offers the necessary API to create and
manage stationary and mobile aglets. Some of the relevant methods in use by us are `createAglet()`, `onCreation()`, `run()`, `dispatch()` etc. In order to create a stationary aglet we first need to create a class to extend the `Aglet` class. This class has all the programming logic for the stationary agent. As an example, the client side has a `RequestProcessor` class to implement the request processor block. This class extends the `Aglet` class and override the methods `onCreation()` and `run()`. The prototype for the `RequestProcessor` class looks as follows:

```java
public class RequestProcessor extends Aglet {
    public void onCreation(Object args){
        //initializes the aglet on creation
        ...
    }

    public void run(){
        //code body for the stationary agent
        ...
    }
}
```

As can be seen, the methods `onCreation()` and `run()` need to be overridden as these methods get implicitly invoked during aglet creation. The aglet can now be created by invoking the following method:

```
createAglet("atp://perth.dbcenter.cise.ufl.edu:434",
    "file:/D/JAVA-APPS/AGLETS1.0.3/PUBLIC/",
    "RequestProcessor.class",
    args);
```

location where the aglet is to be created, the code base of the aglet (as an URL), the name of aglet class whose instance is getting created and finally the arguments for the aglet.
The method creates an instance of the aglet and a new thread is spawned off for this instance. As soon as this thread is created, the `onCreation()` method is called by the aglet to initialize itself. After the initialization, the `run()` method is called. The `run()` method has the actual body of logic for the stationary agent.

**Mobile Aglets:** Now we explain the details for mobile agents. Our system implements a master-slave mechanism of agents. The master agent is a stationary agent. It spawns off one or more slave agents to accomplish a task. The slave agents may or may not be mobile agents depending on whether they need to execute on a remote location or not. A stationary agent can dispatch multiple mobile agents to different locations, which run independently and inform their master on their return. The main reason for this is to parallelize the high latency information retrieval across the various sources in the network, while not tying the master down. The master can retract and dispose agents anytime during processing.

In order to create a slave aglet we have created an abstract class called `Slave`. This abstract class extends the `Aglet` class. The following is the prototype of the `Slave` class:
The mobile slave agent keeps information about the destination site on which it is
going to run on and the aglet information of the master who created it. The abstract class
above provides us a simple platform to generically define and create slave agents. As an
example the shopping agent class on the client side can be defined as follows:

```java
public abstract class Slave extends Aglet {
    URL destination; //URL of the destination
    Aglet Master;     //reference of it's master

    public void onCreation(Object args) {
        //initializes the slave aglet on creation
        initializeTask();
        //instructions about the tasks the aglet
        //should perform on arrival at the destination
        addMobilityListener();
        //dispatch the aglet
        dispatch(destination);
        ...
    }

    abstract void initializeTask() {
        //initialize the slave aglet
        ...
    }

    abstract void doTask() {
        //task to perform
        ...
    }
}

public class ShoppingAgent extends Slave {
    public void initializeTask(Object args) {
        //initializes the slave
        ...
    }

    public void doTask() {
        //code body for the slave
        ...
    }
}
```
A shopping agent can now be created in a manner similar to above i.e.

```java
createAglet("http://perth.dbcenter.cise.ufl.edu:434",
    "file:/D/JAVA-APPS/AGLETS1.0.3/PUBLIC/",
    "ShoppingAgent.class",
    args);
```

This statement spawns of an instance of the slave shopping agent. A separate thread and a new context is provided to this slave. As soon as the slave thread comes alive, it calls the `onCreation()` method. The entire execution life cycle of the slave method is taken care of in this method. It first calls the `initializeTask()` method to initialize the slave, where the slave registers its final destination and stores the information about its master aglet.

The next step is to add a mobility listener. In simple terms this step defines the sequence of execution of the slave when it reaches it's destination. This sequence is explained as follows. The first goal at the destination is for the slave to accomplish the desired task by calling `doTask()`. Then the slave sends back a message to the master accompanied with data if needed. The last step for the slave is to dispose itself off at the remote site itself.

Having defined the tasks to execute at the remote site, the next obvious step is for the slave to dispatch itself to the remote location and execute these tasks. It does this by calling the `dispatch()` method. When the `dispatch()` method is invoked, the slave agent suspends execution. It is then serialized, encoded and transported to the destination. Note that the state and the environment of the aglet are also packaged along with the serialized code. On reaching the destination, the agent code is decoded, deserialized and
prepares to resume execution. The state and environment is restored before the agent starts executing.

The mobile agent can execute on the destination site only if its class definition is available on that site and, if it can be identified by its full class name and discriminator. In case the code is not present at the destination, then it can be sent from the source. The class could also be placed on third site, from where it is then transported to the destination on request. However in both these cases, the class could be transported multiple times, leading to increased network traffic and wasted bandwidth. Additionally, it is recommended that the required class files not be embedded in Java archive files (JAR), because all the classes in the archive are transported to the destination every time an aglet moves. The same argument holds good for any objects created by the mobile agents in process of its execution at the destination.

4.3 Client Side Implementation

The GUI on the client side that interfaces with the user is a straightforward Java applet. The user enters his requirements and requests the results. This causes the applet to create a request processor aglet to exclusively handle the user request from start to finish. This is a stationary aglet and is created in the same manner as is discussed in the previous section.

The request processor’s first task is to capture the user requirement from the applet and translate it to an XML format, appending the appropriate tags to the constraints. The global DTD is referenced to get the right tags.

As an example, let us say that the user is requesting for books and the information entered by the user is as follows:
The priority column indicates that price is most important to the consumer and the days is the least important. The request processor translates these requirements into the following XML document:

```xml
<bookOrders>
  <book>
    <isbn>XYZ</isbn>
    <price>20</price>
    <quantity>1000</quantity>
    <days>4</days>
    <pricePriority>1</pricePriority>
    <quantPriority>2</quantPriority>
    <daysPriority>3</daysPriority>
    . . . . .
    . . . . .
  </book>
</bookOrders>
```

Depending on the item being requested, the request processor comes up with a list of relevant server market sites that hold relevant information on the requested product (in our case, books). For each of these sites it creates a slave shopping agent. The information that the master request processor aglet passes onto each slave Shopping agent is the destination URL to where the slave should be dispatched, the reference to the master aglet's information for messaging purposes and the XML document containing the user request. The slave shopping agent dispatches itself to the specified destination, from where it initiates data retrieval satisfying the request. Once it receives all the data from the market site, it sends it to the master request processor aglet, using the aglets messaging system. The type of the message is "SearchResults". It then dies while still at the server site.
The request processor receives the search results from each of the slave shopping agent aglets; the results still need to be converted to an HTML format so that they can be displayed to the consumer. A stylesheet is defined to perform the conversion. The XSL processor takes this XML document and the stylesheet as inputs and provides the output in the desired HTML file.

4.4 Server Side Implementation

The server side has a stationary aglet called the receiving agent. This master aglet is created right from when the server starts up and is always around. Its main role is to receive messages with the incoming shopping agent aglets from the client’s. This master receiving agent aglet subscribes to a message of type "ShoppingAgent."

When a shopping agent arrives at the market site, it sends a message to multicast its arrival. When the message sender aglet knows the proxy or identity of the receiver, peer-to-peer messaging works fine. When aglets are not aware of the identities of other aglets in a context, they can multicast messages. The context supports message multicasting within a single context. Message multicasting provides a powerful way for aglets to interact and collaborate. The aglets in the context need to subscribe to multicast messages and implement handlers for these messages. In our case, the message is of type "ShoppingAgent" and contains the user request XML document as data. Since the receiving agent subscribes to this type of messages, it receives the message and extracts the XML document from them. It then creates a slave searching agent aglet to actually process this XML request. The searching agent is given the XML document and the reference of the incoming shopping agent. At this point the receiving agent is done with its job, and goes back to receive other shopping agents. The searching agent is
responsible for processing the XML request and providing the corresponding results to the shopping agent.

The XML document which has been received from the shopping agent needs to be converted to a format consistent with the schema at the server side. The searching agent gets the XML request document converted using the XML format converter. As discussed Chapters 2 and 3, there are two XSL documents at each server component, that define two sets of rules of conversion. One, to convert from consumer request to the local format and the other, vice versa. The XML request document and the corresponding XSL document are passed on to the XML converter.

The XML format converter is a stationary aglet that invokes an XSL processor. The XSL processor we use is Xalan-J version 1.0.1. Details of the conversion are explained in the next section. After the XML conversion the XML document from before looks like this,

```
<newRequest>
  <book>
    <isbn>XYZ</isbn>
    <cost>20</cost>
    <number_Reqd>1000</number_Reqd>
    <days_to_deliever>4</days_to_deliever>
    <price_priority>1</price_priority>
    <quant_priority>2</quant_priority>
    <days_priority>3</days_priority>
    . . . . .
  </book>
</newRequest>
```

If we compare the XML document that was created at the consumer site, discussed in section 4.3, and the new XML document created after conversion, as shown above, we note that the data is preserved in the new XML format, but the tags have
changed. The new tags correspond to the schema followed at the particular market site. Also if any changes to the data format were required, that will reflect in the new XML document. For example, if the currency followed at the consumer site was US dollars and the currency at the server site was Canadian dollars, US to Canadian dollar conversion would be carried out during the XML format conversion. This needs to be explicitly defined in the XSL document that defines the rules for conversion. The exchange of information between the searching agent aglet and the XML format converter is by using messages.

The searching agent now has the consumer’s requests in a format that the database can follow, it still has to extract that information before it can send it across to the database processor. To accomplish this, Oracle’s Java based XML parser was used. It is a DOM parser i.e., the XML file to be parsed is converted to a DOM tree that resides in main memory. The key statements that carry out the parsing and return a reference to the DOM tree are listed below.

```java
DOMParser parser = new DOMParser();
parser.parse(url);
XMLDocument doc = parser.getDocument();
Element root = doc.getDocumentElement();
```

`DOMParser()` creates a new parser object and `parse(URL)` parses the XML document pointed to by the given URL. `getDocument()` returns the document that was just parsed, in our case the DOM tree. `getElement()` returns an attribute that allows direct access to the root element of the document.

A pre-order traversal of the DOM tree returns the elements of the XML Document in their order of appearance. The tag the defined the element in the XML
document is the parent node of every element. Hence, all the elements can be identified by their tags. These constraints are sent to the database processor for it to lookup the database for suppliers that would satisfy the consumer’s request.

The results received from the database processor are in the XML format that follows the database schema at the market site. This has to be converted back to the format at the client side. The above conversion process is repeated again.

The searching agent has a reference to the aglet information of the shopping agent. It sends a message of type “results” to the waiting shopping agent with the XML results as part of the message data. When the shopping agent receives this information, it understands that its task at the market site is accomplished. It sends the results back to the master request processor aglet. It then disposes itself.

4.5 XML Conversion

Data conversion is an important step in our implementation. There are conversions between one XML format to another and also from XML to HTML. The details on how these conversions are accomplished are discussed now.

As stated earlier, the main conversions are carried out by the XML format converter that converts from a client XML format to the local server XML format and vice versa. The XML format converter is a stand-alone stationary aglet, although it could simply be a Java program. Two XSL stylesheets are defined for the conversion between the two formats.

The searching agent aglet invokes the XML format converter and specifies the type of conversion it needs. The actual conversion is then carried out by an XSL processor. The XSL processor used in our implementation is the Xalan-J version 1.0.1. As
discussed in Chapter 2, Xalan is an XSL processor for transforming XML documents into HTML, text, or other XML document types. It provides high-performance XSLT stylesheet processing. The statements in the XML format converter aglet that perform the transformations are

```java
XSLTProcessor processor=XSLTProcessorFactory.getProcessor();
processor.process(input, XSLStyleSheet, output);
```

The first statement manufactures the processor for performing transformations. `getProcessor()` gets a new XSLTProcessor with the default high-performance document table model liaison [Xal00b] and XML parser. In the next statement, this XSLTProcessor uses the specified XSLStyleSheet to perform the conversion.

As an example let us consider a customer wishing to buy a book has his requirements requirements in an XML File, `customerOrder.xml`.

```xml
<?xml version="1.0"?>
<book>
   <name>ABC</name>
   <author>J. Sivan</author>
   <quantity>2</quantity>
</book>
```

This XML file reaches the dealer site, but the schema followed is different. Let us say that the schema followed at the dealer site is [title, author numRequired]. A XSL stylesheet `books.xsl` needs to be defined to convert `customerOrder.xml` to `supplier.xml`. The global DTD discussed in Chapter 3 plays a role in defining this stylesheet. The XML document created at the client site follows the DTD, thus the server component knows the format of the incoming document. Knowing the format of the
incoming document and that of the required output, it is fairly simple to construct the XSL document. Listed below is books.xsl.

```xml
<?xml version="1.0"mvc-1.0"?><xsl:stylesheet
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    version="1.0">
    <xsl:template match="book">
        <bookOrder>
            <xsl:apply-templates />
        </bookOrder>
    </xsl:template>
    <xsl:template match="name">
        <title>
            <xsl:value-of select="." />
        </title>
    </xsl:template>
    <xsl:template match="author">
        <author>
            <xsl:value-of select="." />
        </author>
    </xsl:template>
    <xsl:template match="quantity">
        <numRequired>
            <xsl:value-of select="." />
        </numRequired>
    </xsl:template>
</xsl:stylesheet>
```

The original XML file (customerOrder.xml) and the style sheet (books.xsl) are given to the XSL processor resulting in the output XML file (supplier.xml). The statement that does the conversion is

```java
processor.process(customerOrder.xml, books.xsl, supplier.xml);
```

and the converted output file is,
Thus the output file has the necessary tags changes to conform with the local server schema. Note, if there are any changes made to the database schema at the dealer site, only the corresponding literal that forms the tag in the stylesheet need to reflect this.

XML documents to HTML document conversions are similar. This conversion takes place at the client site. When the request processor receives all the search results, they need to be converted to an HTML format to display it to the consumer. Another stylesheet can be defined to perform this conversion.

4.6 Database Query Generation and Relaxation

Query Generation

The searching agent parses the XML document, extracts the consumer’s requirements and hands them over to the database processor. The main role of the database processor is then to frame the query and keep querying the database, while relaxing the constraints, until the required results are returned or till the database is exhausted. The database processor is a stationary aglet. This section describes our implementation of this block along with some ideas on query relaxation.

As soon as the database processor aglet receives the user constraints it goes on to build a SQL query in a StringBuffer object. Let us assume a sample set of constraints as is follows:

```xml
<?xml version="1.0"?>
<bookOrder>
    <title>ABC</title>
    <author>J. Sivan</author>
    <numRequired>2</numRequired>
</bookOrder>
```
All fields marked with an asterisk (*) are mandatory. For all other fields, if the consumer does not state a value, defaults are assumed. For the maximum/minimum conditions, the maximum/minimum values for the respective attributes in the database are assumed. The least price and least number of days are assumed if price and days are not stated. In the above case, the consumer has stated that he requires his books from single suppliers and he needs six such suppliers. The other situation is where the consumer may accept the requirement being satisfied by a combination of sellers. This case is discussed later in the section.

In the current scenario, first the database is queried for six suppliers who can satisfy his requirements. If there are less than the required number of suppliers, the requirements are relaxed and the database is queried again. This process continues until six suppliers are found or until the user's boundary conditions are reached. The pseudocode for the database controller roughly looks like this:

```
*ISBN        XYZ
Price        $15
*Quantity    1000
Days to deliver 4
MaxPrice     $20
MinQuantity  500
MaxDays to deliver 7
*Priority    1-Price, 2-Quantity, 3-Days
Number of options 6
*Multiple Sellers No
```
As a first step, the query representing the initial constraints is framed in a Java StringBuffer object. Access to the database is provided using JDBC. JDBC provides database access via Java that is independent of both the platform and the database host system on which the application runs. The JDBC API defines classes to represent constructs such as database connections, SQL statements and result sets. A database may directly provide a JDBC enabled driver; otherwise a JDBC-ODBC bridge driver is used.

Briefly, the key sequence of statements needed to establish connection and query the database are as follows:

query the database to check the availability of the book
If book is not available
    inform the consumer that the book is unavailable.
else
{
    query the database to check if the required number of options are met when applying the initial constraints
    If required number of options are available
        return options to the user.
    else
    {
        while (options available < required number and max conditions have not yet been reached)
            relax the constraints and check number of options

        if the max conditions stated are reached, and required number of options are not available
            return the currently available set of options

        If(options available > required number)
            Select the best among the currently available set and return to the user.
    }
}
The first statement selects and loads the JDBC-ODBC bridge driver. The second statement establishes connection with the database by getting an instance of the class `Connection`. This instance is needed in all subsequent database accesses. The last two statements execute a query and obtain the results as an instance of the class `ResultSet`.

The `ResultSet` object, `resultSet`, points to one row of the result. Subsequent rows are obtained by repeatedly invoking the `next()` method. Within a row, the column values are extracted by any of the `getXXXX()` methods, like `getString("number")`, etc.

**Query Relaxation**

An important feature of our prototype e-market place system is its ability to relax the user’s constraints in an attempt to provide the best results. Let us walk through some of the ideas we use to achieve this. Although this section does not provide an exhaustive strategy or a specific algorithm, it does provide some thoughts on the relaxation process.

In our example above on books, the consumer is willing to pay $15, wants 1000 books and requires it in 4 days. But he is willing to consider results where the constraints of price could up to $20, number of books down to 500 and the number of days he’s ready to wait for the consignment up to 7. The number of options he requires is 6 i.e., he wants to get 6 suppliers who can satisfy his request in whole. Constraints [15, 1000, 4]
which are [price, quantity, days] respectively are referred to as requested or desired values and [20, 500, 7] which are [maximum price, minimum quantity, maximum days] are the boundary values.

The database is first queried to satisfy the initial constraint of finding six suppliers. If this query does not return six entries, the query needs to be relaxed. The user input includes specifying priorities. In this case price is the first priority, quantity the second and days to deliver the third. This really means when it comes time to relax constraints we start first with days and then move towards price. The algorithm for relaxation is as follows:

```plaintext
for x relaxations of only the third priority attribute
  if (the number of suppliers are met) or
      (all the boundary conditions are reached)
    stop and return the results
    break

for y relaxations of only the second priority attribute
for x relaxations of only the third priority attribute
  if (the number of suppliers are met) or
      (all the boundary conditions are reached)
    stop and return the results
    break

for z relaxations of only the first priority attribute
for y relaxations of only the second priority attribute
for x relaxations of only the third priority attribute
  if (the number of suppliers are met) or
      (all the boundary conditions are reached)
    stop and return the results
    break
```

The efficient implementation of relaxation is to find the right values for x, y and z along with the quantity by which each of the attribute values need to be relaxed each
time. The attribute days is always incremented in steps of one while price and quantity are incremented by a percentage difference between the requested and the maximum units.

Let’s explain the algorithm with our example. In case the query with the original constraints returns fewer than required suppliers, we first try to relax the least priority attribute i.e. delivery days. We do this $x$ times, checking to see if the constraints are met each time. If after $x$ relaxations of days, the request is still not satisfied, we proceed to relax the next least priority attribute i.e. quantity. We relax quantity once, if the request is not satisfied, we again do the relaxation of delivery days $x$ times. We continue doing this combination of increasing quantity and delivery days a total of $y$ times. Even after this if further relaxation is needed, then we relax price. If at anytime during the relaxation, more options, $n$, are available than are required, then the best or closest $n$ values are considered by applying the geometric distance-formula.

Let us now see how the relaxation process functions in our example. For simplicity, let’s just consider two constraints i.e. the price and delivery days. In figure 4.1 each dot on the graph represents a supplier. The cost of the book is on the Y-axis, while delivery days is on the X-axis. The position of any dot D(b,a) on the graph, indicates that the supplier D sells the book at 'a' dollars and the days he takes to deliver them is 'b'.

Our first step is to run the query with the desired constraints of [15, 4] or [B0,A0] for price and days. This is represented by the dotted box Io. As we can see, only two suppliers are encompassed in that box. We still need four more suppliers, so we begin to relax the query. Since days has the least importance on the user's priority, we relax days by 1 to the point B1. The result domain of this relaxation now includes box I1. There are
no suppliers in this box, so we proceed to further relax the delivery days constraint by 1 to the point B2. Looking into the box I2, we see one more supplier, but we still need three more.

![Figure 4.1: Sample price vs days distribution](image)

We need to continue relaxing our constraints. And we have already relaxed delivery days twice, so now we try to relax the next least important constraint which is price. Let's say we increase the price by 20% to the point A1, so as to include box I3. This gives us two more suppliers, making the total to five. At this point it may be interesting to add that, just in case we were looking only for four suppliers, we need to
make a choice between the two suppliers from box I3. The solution is simple, we simply apply the distance formula and choose the supplier closer to point A0.

Returning back to our example, we still are in need of one more supplier. Now, we have already relaxed delivery days twice and price once, so let's just increase delivery days again by 1 to the point B3. This includes the box I4. Again, we have two suppliers here and we need only one. So by the argument posed above, we choose the one closer to B2.

We have stopped our relaxation process here, since we found six suppliers. However in case we did not, then we would proceed in a similar way, by also including the third constraint of number of books, making it a three dimensional picture.

By looking at the suppliers in the database who stocked the required book and the set of suppliers returned to the consumer after performing the required relaxation, we see, that the relaxation algorithm has been effective.

**Multiple Seller Option**

In the previous example the user explicitly states that he requires the entire order from a single supplier and request for the six best supplier options. But he could also choose to receive the requested quantity either from a single supplier or from a combination of suppliers. The latter case comes in handy when the consumer requires a large quantity and any one supplier may not be able to satisfy his request.

This case is dealt with in a similar manner, however, only price and delivery days would be relaxed and not quantity. The goal is to relax each attribute and look at the suppliers until the total quantity is met. As soon as the quantity is satisfied, we return all the suppliers to the user.
4.7 Conclusion

In this chapter we have given a detailed outline of the implementation of our electronic market place. The aglet API offers an abstract class on which our stationary and remote aglets are defined. The shopping agent which is a mobile agent defines its task at the client side, before dispatching itself. It does this by means of a mobility listener, this enables the agent to execute the actions as soon as it reaches the market site. The Searching agent at the market site is a slave agent that processes the entire request for the Shopping agent. XML conversion is mandatory for bringing together diversely defined, yet similar systems and needs an XSL processor along with some stylesheet definitions. Xalan-J was our XSL processor. Finally we put forth some ideas on query relaxation, such that the user request is always satisfied in the best possible way. Our algorithm takes into consideration the priorities of the constraints and relaxes them accordingly. Use of the algorithm on the market site saves the consumer the trouble of initiating a new search for every time that there were no matches returned for a set of constraints that he had stated.
CHAPTER 5
EXPERIMENTAL RESULTS

In the earlier chapters, we have discussed the design and architecture of our intelligent web-based market place. In this chapter, we discuss the underlying infrastructure on which we have implemented and installed our prototype e-market place system, as well as the test procedures used for its analysis.

5.1 Software Overview

In our implementation, we created an e-market place that sells books. It consists of three market sites running the aglet server called Tahiti [13]. Apart from providing the aglets an environment to operate in, Tahiti also provides a user interface for monitoring, creating, dispatching, and disposing of agents and also for setting the agent access privileges for the aglet server. The database used at each market site was Microsoft Access. The consumer site was interfaced with an Internet browser. The consumer site needed to be agent enabled because the information from the client site is sent to the market sites using mobile agents. Each of these sites, both server and client, was built on a Pentium II 400 MHz with 128 MB RAM computer terminal. The platform was Windows NT. Communication protocol used by the aglets was TCP/IP

5.2 The Test Database

The database at each market site had information about the products being sold, suppliers registered at the site, etc. Though databases of different market sites follow
different schemas, all of them had similar information stored in them. This includes information about the products, suppliers, consumers and also negotiation information.

As this thesis concentrates on the first phase of operations, namely, the pre-selection phase, the first three categories of information are of concern to us. Product information contains details about the product, suppliers who stock the particular product, price they quote, etc. Supplier information stores information about all the suppliers registered with the market site, including supplier name, address, etc., and his/her credibility. Information about the supplier’s credibility is basically a consumer rated attribute. Consumers rate the suppliers based on factors such as on-time delivery, quality of goods, etc. This rating comes in handy when a consumer needs to pick a few suppliers from a large set, where all suppliers closely match his request. A supplier’s credibility could be the deciding factor. This information is also used by the request processor on the client site. For instance, if the consumer had stated that he requires $x$ supplier options and number of results returned to the request processor is more than $x$, it needs to choose the best $x$ options. In many situations, most of the suppliers would have a similar quote. It is fairly difficult to choose one supplier over the other in such cases. One way to select suppliers could be based on their credibility. If all suppliers are equal on their credibility rating and offer the same quote to the supplier, then, all of them are presented to the consumer. It is now left to the consumer to take a decision.

Consumer information stores all information pertaining to the consumers. This includes the consumer’s name, address, frequently purchased product and his credibility. Similar to how a supplier’s credibility rating could influence the consumer’s decision, the consumer’s credibility rating may influence the suppliers negotiation strategy or in an
extreme case, a supplier may not want to negotiate or do business with a consumer whose credibility is low. Suppliers rate their consumers. The ratings could depend on how prompt the consumer is in making payments, etc.

Though we do not use negotiation information in the first phase, we will briefly explain how it comes in handy in the negotiation phase. The negotiation strategies of suppliers are stored in the database at the market site and are available at all times. The suppliers do not have to remain online 24 hours of the day to respond to negotiation requests. If a supplier is offline, the supplier negotiation agent, discussed in Chapter 1, requests the database controller to retrieve the negotiation strategy of a supplier from the database. The market floor supplier negotiators, also discussed in Chapter 1, are then given this information to carry out negotiations.

As stated earlier the database we use is Microsoft Access. Every market site has its database. There are currently about 20-30 suppliers registered with each market site. The market place sells about 30-40 books. Hence for each book, there are about 30 different quotes to choose from. To facilitate rigorous testing, 2-3 books had over a hundred different quotes.

To populate the database, we looked up the Internet for databases but found that none followed our schema. The data was generated by a C program that we wrote and stored in a text file. It was then imported into the database.

5.3 Test Suite

In our e-market place, the consumer enters information about the book he requires--the ISBN or the title, the price, quantity, days, single/multiple suppliers, number of suppliers,
priorities, etc. He is returned a list of suppliers who can deliver the required number of books, for the price he quoted, within the stated number of days.

To understand how the set of suppliers is chosen, it is best to execute queries with different sets of constraints and analyze the results returned. Let us now step through some tests for a book with ISBN 16. There were nine suppliers who stocked ISBN 16, at a particular market site. The relevant database entries viz. the price quoted, number of books in stock and number of days for delivery, for each of these nine suppliers are listed in Table 5-1 below. The priorities stated were in the order, price, quantity and days, with price having the highest priority.

<table>
<thead>
<tr>
<th>Supplier_id</th>
<th>Price</th>
<th>Quantity</th>
<th>Days</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>169</td>
<td>184</td>
<td>9</td>
</tr>
<tr>
<td>S2</td>
<td>165</td>
<td>68</td>
<td>2</td>
</tr>
<tr>
<td>S3</td>
<td>169</td>
<td>22</td>
<td>12</td>
</tr>
<tr>
<td>S4</td>
<td>168</td>
<td>132</td>
<td>13</td>
</tr>
<tr>
<td>S5</td>
<td>161</td>
<td>77</td>
<td>7</td>
</tr>
<tr>
<td>S6</td>
<td>164</td>
<td>9</td>
<td>11</td>
</tr>
<tr>
<td>S7</td>
<td>164</td>
<td>302</td>
<td>8</td>
</tr>
<tr>
<td>S8</td>
<td>164</td>
<td>486</td>
<td>10</td>
</tr>
<tr>
<td>S9</td>
<td>164</td>
<td>486</td>
<td>10</td>
</tr>
<tr>
<td>S10</td>
<td>164</td>
<td>486</td>
<td>10</td>
</tr>
</tbody>
</table>
Table 5-2: Constraint sets and the respective results.

<table>
<thead>
<tr>
<th>Set #</th>
<th>Price (P)</th>
<th>Max Price</th>
<th>Qty (Q)</th>
<th>Min Qty</th>
<th>Days (D)</th>
<th>Max Days</th>
<th>Supplier</th>
<th>Results (with relaxation)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>60</td>
<td>100</td>
<td>100</td>
<td>-</td>
<td>3</td>
<td>-</td>
<td>2</td>
<td>Min P for book is 161</td>
</tr>
<tr>
<td>2</td>
<td>75</td>
<td>-</td>
<td>1000</td>
<td>600</td>
<td>3</td>
<td>-</td>
<td>4</td>
<td>Max Q is 486</td>
</tr>
<tr>
<td>3</td>
<td>100</td>
<td>-</td>
<td>1000</td>
<td>-</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>Min D to deliver is 2</td>
</tr>
<tr>
<td>4</td>
<td>160</td>
<td>165</td>
<td>30</td>
<td>-</td>
<td>4</td>
<td>-</td>
<td>3</td>
<td>S2, S6, S7</td>
</tr>
<tr>
<td>5</td>
<td>160</td>
<td>165</td>
<td>30</td>
<td>-</td>
<td>4</td>
<td>4</td>
<td>3</td>
<td>S2, S6</td>
</tr>
<tr>
<td>6</td>
<td>160</td>
<td>165</td>
<td>1000</td>
<td>-</td>
<td>4</td>
<td>-</td>
<td>3</td>
<td>S6, S9, S10</td>
</tr>
<tr>
<td>7</td>
<td>160</td>
<td>-</td>
<td>1000</td>
<td>-</td>
<td>4</td>
<td>-</td>
<td>3</td>
<td>S1, S9, S10</td>
</tr>
<tr>
<td>8</td>
<td>165</td>
<td>-</td>
<td>10</td>
<td>-</td>
<td>5</td>
<td>-</td>
<td>1</td>
<td>S6</td>
</tr>
<tr>
<td>9</td>
<td>165</td>
<td>-</td>
<td>1000</td>
<td>-</td>
<td>5</td>
<td>-</td>
<td>M</td>
<td>S6, S7, S9, S10</td>
</tr>
<tr>
<td>10</td>
<td>160</td>
<td>-</td>
<td>10000</td>
<td>-</td>
<td>5</td>
<td>-</td>
<td>M</td>
<td>All. But, max Q is 1441</td>
</tr>
</tbody>
</table>

Table 5-2 above lists the different constraint sets and their respective results. The columns price, quantity and days contain the price the consumer was willing to pay, the quantity he desired and the number of days he required his books in. The columns maxPrice, minQuant and maxDays contain the boundary values for the price, quantity and delivery date respectively. The supplier column indicates if the consumer wanted his books from a combination of suppliers (M) or a single supplier. In the latter case, the column states the number of supplier options the consumer had requested. The results column states the result returned for each constraint set. We will now analyze the results.

Set 1 is fairly simple. The user requires 100 books in 3 days and is ready to pay $100 a book. A message “Minimum price for ISBN 16 is $161” was received from the market site. This is because the minimum price that ISBN 16 was available for, at the market site, was $161, where as the maximum price stated by the consumer was $100.

For set 4, the three suppliers selected were S2, S6 and S7. As stated in an earlier chapter, a ‘strict’ ordering policy that adheres to the priority on a certain attribute would return suppliers in an ascending or descending list ordered on that attribute. In the case
mentioned above, as price is the attribute with the highest priority and therefore all suppliers would be returned in the order of the price they were quoting regardless of their quotes on other attributes. The selection strategy that has been implemented however does not use a strict ordering policy. It utilizes a priority-weighted combination of all other attributes as well. The resulting list of suppliers is therefore not in a strict price order, as, the relaxation mechanism listed suppliers in order of their net weighted value of their quotes.

The price quotes of suppliers S2, S6, S7, S8, S9 and S10, fall in the requested price range. S6 has the best net weight, and hence, was chosen. S7 offers the least price, satisfies the quantity requirement. Days to deliver is 7, which is greater than the user’s desired requirement, but there is no maximum value stated. Hence, S7 has a better net weight than the remaining. Among S2, S9 and S10, though S2 has a quote that is $1 more than the latter two, but books are delivered in 2 days, compared to 8 and 10 by S9 and S10 respectively. Hence, S2 is rated better overall and was selected.

Set 5 is similar to set 4 expect for the maximum number of days constraint which was stated as 4. Suppliers S2 and S6 were the only suppliers returned, even though 3 suppliers were requested. This is because they are the only two suppliers that can deliver in 4 or less days. For set 8, all suppliers qualify to be selected, as there are no maximum/minimum values stated. S6 was chosen out of the nine, as it had the best net weighted value. S7 quotes a price lesser than S6, but delivers the books in 7 days, which results in a net weight that looses out to S6’s net weight. S2 delivers the books in 2 days, but with days to deliver being the third priority, this is given less weightage compared to price. S6 has a lower price quote than S2 and is hence chosen.
Sets 9 and 10 have the multiple seller constraint. As discussed in Chapter 4, a consumer may choose to receive his order from a set of suppliers, especially if he/she requires a large number of books. S6, S7, S9 and S10 were returned for constraint set 9. Price was stated to be the first priority, hence the suppliers were selected in increasing order of their quotes. Also, there was no limit to the number of suppliers.

Thus far we have seen how constraint relaxation works using the net weight approach. The interesting thing to note is that the algorithm we have used essentially simulates intelligence, built on a priority based weighing formula and closely matches real-life solutions.

To rigorously test our relaxation algorithm, we generated a number of consumer requests. Having created the database, we had a good idea about the data it contained. We checked for a number of conditions;

- we started with the simplest cases, where the database had at least as many suppliers who could satisfy the consumer’s constraints, as required i.e., there was no relaxation required to retrieve the required number of suppliers

- we carried out a number of boundary condition checks; for example, where the consumer’s price requirement was lower than the cheapest quote available at the market site

- we carried out extensive tests to check for conditions where various attributes needed to be relaxed to get the required number of results. This included relaxing just one attribute, to relaxing all three. These tests were run for different combinations of priorities
• we also executed tests where relaxation did not yield the required number of results. This was for one of two reasons: the boundary conditions were strict, resulting in a restricted amount of relaxation; the database did not carry the required number of options

• we tested for the multiple seller options. If the required quantity was available, a list of suppliers who satisfied the request was returned, in the order of priorities stated.

In all cases, a message was sent to the consumer summarizing the results of the search. If the search was successful, it stated so. The list of suppliers was also displayed to the user. If the required number of options was unavailable, the message stated so. It also indicated the steps that were taken to arrive at the required number, why it failed, and what the consumer needs to do if he does need all the options. A typical message read, “Attributes were relaxed in the order stated, but the search did not yield the requested number of suppliers. Please relax the upper/lower boundaries of the attributes and try again”. It is now up to the consumer to relax constraints and try again. In certain other cases, where the maximum conditions stated were lower than the lowest in the database, the consumer was explicitly told that. The message would then read “Your quote for price is $400, the minimum price that the product is available for is $500. Please change your desired price and try again.”

5.4 Conclusions

We have discussed our test bed and testing procedures so far. We now state some conclusions we have arrived at about the performance of our market place. We start with software agents. As discussed in Chapter 3, there is one agent sent to each market site that needs to be explored for lists of suppliers. Initially, we created and dispatched just
one agent. This agent was packed with addresses of all the sites that needed to be visited. At the end of its itinerary, it would return the results to the consumer site. The response time in this case was the sum of the response times at each of the sites. We modified this agent, making it return the results obtained at a particular site as soon as it received it. Though this method allowed the consumer to receive intermediate results, results from the last set of market sites, still took as much time as the previous case. To improve the response time, we decided to search the market sites concurrently, by dispatching an agent to each site. This process of parallellizing the search certainly improves the response time. The only additional work done is at the client site, where the request processor needs to create a number of agents and integrate the results as they come in. But this is CPU intensive task and is negligible compared to the network related latencies.

The relaxation algorithm followed at the market sites saves the consumer the trouble of reentering constraints, if there were no matches found. It returns the next best or closest results that satisfy his requirements. Our relaxation algorithm works well when the constraints are prioritized. More sophistication needs to be added. Relaxation should be carried out even if no priorities are stated and even if two or more constraints have the same priority. Unlike the CoBase system we discussed in Chapter 2, there were no abstractions that were pre-defined. All the relaxation was performed dynamically.

The market place also provides the consumer the ‘multiple seller option.’ This comes in handy when the consumer requires a large quantity and is not too concerned about the number of suppliers or the combination of suppliers, who together can satisfy
his request. Here too, priorities are taken into consideration. Suppliers are selected in the order of priorities stated.
6.1 Summary

Most web-based trading centers of today are e-commerce sites that allow a consumer the means to look for a product at the particular vendor or retailer site. While this step is a huge leap in the way business was done a few years ago, the consumer would still prefer to look for and compare choices across multiple vendors and retailers. Furthermore, it would be beneficial if the consumer also receives options, which do not exactly match his requirements, but closely match them while taking care of his priorities.

The electronic market place we have proposed in this thesis, endeavors to do just that. It offers the consumer a one-stop shop to search for products across a diverse list of vendors and retailers. To be a part of the system, the vendors do not need to change their infrastructure. They simply register with a market site, providing instant access to their database. Another important aspect of our market place is its ability to search for closest matching requirements in absence of exact matches. Price may not always be the first priority for a consumer. For example a bookstore looking to stock 1000 copies of a much-needed book, may find it more important to find suppliers to stock the requisite number of books within a week.

There needs to be some mechanism where the user’s requirement can be intelligently modified to arrive at a set of results, which would be almost similar to those
if the users spent a few minutes searching himself. Our query relaxation algorithm attempts to fulfill this role. The user is allowed to specify what attributes of his inputs are of higher priority to him i.e. which attribute he is least flexible on. Our relaxation algorithm is iterative, where in each iteration it tries to relax the least priority item more times that of the highest priority one. Some of the test cases displayed in chapter 5, showed the effectiveness of the algorithm in choosing good results. The attributes such as price and quantity are relaxed by a percentage, where as days is relaxed in units of one or two days.

The choice of the model to be chosen for implementing our e-market place, was between a thin-client model and the software agent model. Software agents ship the processing to the data, instead of data to the processing. Thus they tend to reduce network traffic and overcome network latency. When a user enters his request, the client side dispatches a number of mobile agents to all the relevant market sites. The market site is queried for information, along with relaxation of constraints, if necessary.

Aglets were the agent system used in our market place, because of its Java implementation and a powerful Java API that makes deploying stationary and mobile agents quite simple. Mobile aglets have been packed with the intelligence to transfer themselves to a site, initiate a search at the site and return results to the consumer, while stationary agents aid in the actual searching.

With the common platform provided by diverse systems, processing data from heterogeneous sites or sources should be simple. But this is not the case. Validating data format and ensuring content correctness are still major hurdles to achieve simple, automatic exchanges of data. In an e-market place scenario like ours, where there are a
number of vendors who need to exchange information and understand what the other is saying, it is important to follow a foolproof way to do so. XML technology that we have used as the format for data exchange remedies this problem.

In addition, as both XML and Java support Unicode character sets, they support the development of internationalized applications. Using XML markup as the format for data exchange and Java based agents enables our system to be a truly global marketplace.

### 6.2 Contributions

We have created an electronic marketplace system that gives the user instant access to an ever increasing set of suppliers and returns a list of suppliers who match his request. Though there are sites that provide results after searching through a number of possible supplier options, currently, there are no sites that exhaustively search different supplier sites and endeavor to return closest matching results to a user by relaxing his constraints in accordance with his preferences. We have designed an algorithm that does just this. Our dynamic query relaxation algorithm scours the database to find the next best set of matches in the absence of exact matches.

We have also explored the possibilities of using a relatively new technology, XML, as the data exchange format in an electronic trading environment. We were able to appreciate the ease in transferring documents between different sites following different schemas. We have also contributed ideas to introduce negotiations between suppliers and consumers, at the market place.
6.3 Future Work

For any system to use agent software extensively, every node in the system needs to support infrastructure to support the operation of agents. In a market place scenario, the consumer and market sites need agent servers to provide the required environment. Installing servers on the market site may not pose a problem. But, consumers may not want to install a server on their node for various reasons. So, currently we do not know how viable an agent based system would be. There should be a scenario where a consumer or any other entity could reap the benefits of an agent system without having the need to support specialized software.

Additionally, most agent systems differ in architecture and implementation. Hence interoperability is a huge concern. The general acceptance of mobile agents for network management will depend heavily upon standards.

Our query relaxation algorithm is simple and works well when all the constraints are prioritized. More sophistication needs to be added. A consumer may not wish to prioritize his requirements. The relaxation algorithm should be such that it still returns approximate results i.e., it should be knowledge based. To introduce such knowledge, it is necessary to analyze the domain of the required data. When the relaxation is required, this knowledge of domain and data is used to decide the order in which constraints would be relaxed and by how much i.e., generic relaxation steps are defined. For example, let us suppose that a consumer could order flowers and furniture among other products at a particular site. In a general case, the date of delivery is more important in case of flowers than furniture. The algorithm should be intelligent enough to know about the product and decide the attribute for relaxation. In the case of the above example, the algorithm should query for florists who can deliver flowers on the particular day, relaxing the price
constraint if required. This need not be the case with furniture. The constraint of
dimensions or color may be of more importance.

A consumer may also enter an ambiguous set of constraints. The query relaxation
procedure should be able to recognize and tolerate imprecisely specified queries and
return an appropriate set of results. For example, suppose a consumer wishes to purchase
a book, but cannot remember too many details about it. The algorithm should be able to
come up with a set of possible books with the just the information input by the consumer
and should still return a set of results.

Currently, we are relaxing the attributes like price and quantity by a percentage
and days for delivery by a fixed number of days i.e., we are equating 10% or 20% of the
difference in desired price and maximum price, to one or two days. We need to attach
more weight to the attributes and arrive at these units of relaxation empirically to yield
more accurate results.

The introduction of a time-out mechanism would be helpful. With an efficient
time-out mechanism, the request processor at the consumer site need not wait indefinitely
for results from the shopping agents.

The second phase of the market place operation is the negotiation phase. The
components required and the order of flow of data have been defined for this phase.
However, finer design details and negotiation strategies need to be worked on such that
the entire negotiation process could be automated.
REFERENCES


[Chu94] Chu, W., Chen, Q., A structured approach for cooperative query answering, Transactions on Knowledge and Data Engineering, 6(5) : pp 738-749, October 1994.


BIOGRAPHICAL SKETCH

Jagadha Sivan was born on September 5th 1975, in Madras, India. She received a bachelor's degree in information science and engineering securing first class with distinction from Bangalore University, Bangalore, India, in August 1997.

She joined the University of Florida in August 1998 to pursue a master’s degree in the Department of Computer and Information Science and Engineering.

Her research interests include mobile agents and XML.